Optimizing and Incrementalizing Higher-order Collection Queries by AST Transformation

Dissertation

der Mathematisch- und Naturwissenschaftlichen Fakultät

der Eberhard Karls Universität Tübingen

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

vorgelegt von

Paolo G. Giarrusso

aus Catania (Italien)

Tübingen

2018

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	26.01.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Klaus Ostermann
2. Berichterstatter:	Prof. Dr. Fritz Henglein

To the memory of my mother Rosalia (1946–2016)

Alla memoria di mia madre Rosalia (1946–2016)

Abstract

In modern programming languages, queries on in-memory collections are often more expensive than needed. While database queries can be readily optimized, it is often not trivial to use them to express collection queries which employ nested data and first-class functions, as enabled by functional programming languages.

Collection queries can be optimized and incrementalized by hand, but this reduces modularity, and is often too error-prone to be feasible or to enable maintenance of resulting programs. To free programmers from such burdens, in this thesis we study how to optimize and incrementalize such collection queries. Resulting programs are expressed in the same core language, so that they can be subjected to other standard optimizations.

To enable optimizing collection queries which occur inside programs, we develop a staged variant of the Scala collection API that *reifies* queries as ASTs. On top of this interface, we adapt domain-specific optimizations from the fields of programming languages and databases; among others, we rewrite queries to use indexes chosen by programmers. Thanks to the use of indexes we show significant speedups in our experimental evaluation, with an average of 12x and a maximum of 12800x.

To incrementalize higher-order programs by program transformation, we extend *finite differencing* [Paige and Koenig, 1982; Blakeley et al., 1986; Gupta and Mumick, 1999] and develop the first approach to incrementalization by program transformation for higher-order programs. Base programs are transformed to *derivatives*, programs that transform input *changes* to output changes. We prove that our incrementalization approach is correct: We develop the theory underlying incrementalization for simply-typed and untyped λ-calculus, and discuss extensions to System F.

Derivatives often need to reuse results produced by base programs: to enable such reuse, we extend work by Liu and Teitelbaum [1995] to higher-order programs, and develop and prove correct a program transformation, converting higher-order programs to *cache-transfer-style*.

For efficient incrementalization, it is necessary to choose and incrementalize by hand appropriate primitive operations. We incrementalize a significant subset of collection operations and perform case studies, showing order-of-magnitude speedups both in practice and in asymptotic complexity.

Zusammenfassung

In modernen, universellen Programmiersprachen sind Abfragen auf Speicher-basierten Kollektionen oft rechenintensiver als erforderlich. Während Datenbankenabfragen vergleichsweise einfach optimiert werden können, fällt dies bei Speicher-basierten Kollektionen oft schwer, denn universelle Programmiersprachen sind in aller Regel ausdrucksstärker als Datenbanken. Insbesondere unterstützen diese Sprachen meistens verschachtelte, rekursive Datentypen und Funktionen höherer Ordnung.

Kollektionsabfragen können per Hand optimiert und inkrementalisiert werden, jedoch verringert dies häufig die Modularität und ist oft zu fehleranfällig, um realisierbar zu sein oder um Instandhaltung von entstandene Programm zu gewährleisten. Die vorliegende Doktorarbeit demonstriert, wie Abfragen auf Kollektionen systematisch und automatisch optimiert und inkrementalisiert werden können, um Programmierer von dieser Last zu befreien. Die so erzeugten Programme werden in derselben Kernsprache ausgedrückt, um weitere Standardoptimierungen zu ermöglichen.

Teil I entwickelt eine Variante der Scala API für Kollektionen, die *Staging* verwendet um Abfragen als abstrakte Syntaxbäume zu *reifizieren*. Auf Basis dieser Schnittstelle werden anschließend domänenspezifische Optimierungen von Programmiersprachen und Datenbanken angewandt; unter anderem werden Abfragen umgeschrieben, um vom Programmierer ausgewählte Indizes zu benutzen. Dank dieser Indizes kann eine erhebliche Beschleunigung der Ausführungsgeschwindigkeit gezeigt werden; eine experimentelle Auswertung zeigt hierbei Beschleunigungen von durchschnittlich 12x bis zu einem Maximum von 12800x.

Um Programme mit Funktionen höherer Ordnung durch Programmtransformation zu inkrementalisieren, wird in Teil II eine Erweiterung der *Finite-Differenzen-Methode* vorgestellt [Paige and Koenig, 1982; Blakeley et al., 1986; Gupta and Mumick, 1999] und ein erster Ansatz zur Inkrementalisierung durch Programmtransformation für Programme mit Funktionen höherer Ordnung entwickelt. Dabei werden Programme zu *Ableitungen* transformiert, d.h. zu Programmen die *Eingangsdifferenzen* in *Ausgangdifferenzen* umwandeln. Weiterhin werden in den Kapiteln 12–13 die Korrektheit des Inkrementalisierungsansatzes für einfach-getypten und ungetypten λ-Kalkül bewiesen und Erweiterungen zu System F besprochen.

Ableitungen müssen oft Ergebnisse der ursprünglichen Programme wiederverwenden. Um eine solche Wiederverwendung zu ermöglichen, erweitert Kapitel 17 die Arbeit von Liu and Teitelbaum [1995] zu Programmen mit Funktionen höherer Ordnung und entwickeln eine Programmtransformation solcher Programme im *Cache-Transfer-Stil*.

Für eine effiziente Inkrementalisierung ist es weiterhin notwendig, passende Grundoperationen auszuwählen und manuell zu inkrementalisieren. Diese Arbeit deckt einen Großteil der wichtigsten Grundoperationen auf Kollektionen ab. Die Durchführung von Fallstudien zeigt deutliche Laufzeitverbesserungen sowohl in Praxis als auch in der asymptotischen Komplexität.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many program perform queries on collections of data, and for non-trivial amounts of data it is useful to execute the queries efficiently. When the data is updated often enough, it can also be useful to update the results of some queries *incrementally* whenever the input changes, so that up-to-date results are quickly available, even for queries that are expensive to execute.

For instance, a program manipulating anagraphic data about citizens of a country might need to compute statistics on them, such as their average age, and update those statistics when the set of citizens changes.

Traditional relational database management systems (RDBMS) support both queries optimization and (in quite a few cases) incremental update of query results (called there *incremental view maintenance*).

However, often queries are executed on collections of data that are not stored in a database, but in collections manipulated by some program. Moving in-memory data to RDBMSs typically does not improve performance [Stonebraker et al., 2007; Rompf and Amin, 2015], and reusing database optimizers is not trivial.

Moreover, many programming languages are far more expressive than RDBMSs. Typical RDBMS can only manipulate SQL relations, that is multisets (or bags) of tuples (or sometimes sets of tuples, after duplicate elimination). Typical programming languages (PL) support also arbitrarily nested lists and maps of data, and allow programmers to define new data types with few restrictions; a typical PL will also allow a far richer set of operations than SQL.

However, typical PLs do not apply typical database optimizations to collection queries, and if queries are incrementalized, this is often done by hand, even though code implementing incremental query is error-prone and hard-to-maintain [Salvaneschi and Mezini, 2013].

What's worse, some of these manual optimizations are best done over the whole program. Some optimizations only become possible after inlining, which reduces modularity if done by hand.[1]

Worse, adding an index on some collection can speed up looking for some information, but each index must be maintained incrementally when the underlying data changes. Depending on the actual queries and updates performed on the collection, and on how often they happen, it might turn out that updating an index takes more time than the index saves; hence the choice of which indexes to enable depends on the whole program. However, adding/removing an index requires updating all PL queries to use it, while RDBMS queries can use an index transparently.

Overall, manual optimizations are not only effort-intensive and error-prone, but they also significantly reduce modularity.

[1] In Chapter 2 we define modularity as the ability to abstract behavior in a separate function (possibly part of a different module) to enable reuse and improve understandability.

1.1 This thesis

To reduce the need for manual optimizations, in this thesis we propose techniques for optimizing higher-order collection queries and executing them incrementally. By generalizing database-inspired approaches, we provide approaches to query optimization and incrementalization by code transformation to apply to higher-order collection queries including user-defined functions. Instead of building on existing approaches to incrementalization, such as self-adjusting computation [Acar, 2009], we introduce a novel incrementalization approach which enables further transformations on the incrementalization results. This approach is in fact not restricted to collection queries but applicable to other domains, but it requires adaptation for the domain under consideration. Further research is needed, but a number of case studies suggest applicability, even in some scenarios beyond existing techniques [Koch et al., 2016].

We consider the problem for functional programming languages such as Haskell or Scala, and we consider collection queries written using the APIs of their collection libraries, which we treat as an embedded domain-specific language (EDSL). Such APIs (or DSLs) contain powerful operators on collections such as *map*, which are higher-order, that is they take as arguments arbitrary functions in the host language that we must also handle.

Therefore, our optimizations and incrementalizations must handle programs in higher-order EDSLs that can contain arbitrary code in the host language. Hence, many of our optimizations will exploit on properties of our collection EDSL, but will need to handle host language code. We restrict the problem to purely functional programs (without mutation or other side effects, mostly including non-termination), because such programs can be more "declarative" and because avoiding side effects can enable more powerful optimization and simplify the work of the optimizer at the same time.

This thesis is divided into two parts:

- In Part I, we optimize collection queries by static program transformation, based on a set of rewrite rules [Giarrusso et al., 2013].

- In Part II, we incrementalize programs by transforming them to new programs. This thesis presents the first approach that handles higher-order programs through program transformation; hence, while our main examples use collection queries, we phrase the work in terms of λ-calculi with unspecified primitives [Cai, Giarrusso, Rendel, and Ostermann, 2014]. In Chapter 17, we extend ILC with a further program transformation step, so that base programs can store intermediate results and derivatives can reuse them, but without resorting to dynamic memoization and necessarily needing to look results up at runtime. To this end, we build on work by Liu [2000] and extend it to a higher-order, typed setting.

Part II is more theoretical than Part I, because optimizations in Part I are much better understood than our approach to incrementalization.

To incrementalize programs, we are the first to extend to higher-order programs techniques based on finite differencing for queries on collections [Paige and Koenig, 1982] and databases [Blakeley et al., 1986; Gupta and Mumick, 1999]. Incrementalizing by finite differencing is a well-understood technique for database queries. How to generalize it for higher-order programs or beyond databases was less clear, so we spend significant energy on providing sound mathematical foundations for this transformation.

In fact, it took us a while to be sure that our transformation was correct, and to understand why; our first correctness proof [Cai, Giarrusso, Rendel, and Ostermann, 2014], while a significant step, was still more complex than needed. In Part II, especially Chapter 12, we offer a mathematically much simpler proof.

Contributions of Part I are listed in Sec. 2.1. Contributions of Part II are listed at the end of Sec. 10.7.

1.1.1 Included papers

This thesis includes material from joint work with colleagues.

Part I is based on work by Giarrusso, Ostermann, Eichberg, Mitschke, Rendel, and Kästner [2013], and Chapters 2 to 4, 6, 8 and 9 come from that manuscript. While the work was in collaboration, a few clearer responsibilities arose. I did most of the implementation work, and collaborated to the writing: among other things I devised the embedding for collection operations, implemented optimizations and indexing, implemented compilation when interpretation did not achieve sufficient performance, and performed the performance evaluation. Michael Eichberg and Ralf Mitschke contributed to the evaluation by adapting FindBugs queries. Christian Kästner contributed, among other things, to the experiment design for the performance evaluation. Klaus Ostermann proposed the original idea (together with an initial prototype) and supervised the project.

The first chapters of Part II are originally based on work by Cai, Giarrusso, Rendel, and Ostermann [2014], though significantly revised; Chapter 10 contains significantly revised text from that paper, and Chapters 16 and 19 survive mostly unchanged. This work was even more of a team effort. I initiated and led the overall project and came up with the original notion of change structures. Cai Yufei contributed differentiation itself and its first correctness proof. Tillmann Rendel contributed significant simplifications and started the overall mechanization effort. Klaus Ostermann provided senior supervision that proved essential to the project. Chapters 12 and 13 contain a novel correctness proof for simply-typed λ-calculus; for its history and contributions see Sec. 13.5.

Furthermore, Chapter 17 has recently been published in revised form [Giarrusso, Régis-Gianas, and Schuster, 2019]. I designed the overall approach, the transformation and the case study on sequences and nested loops. Proofs were done in collaboration with Yann Régis-Gianas: he is the main author of the Coq mechanization and proof, though I contributed significantly to the correctness proof for ILC, in particular with the proofs described in Appendix C and their partial Agda mechanization. The history of this correctness proof is described in Appendix C.9. Philipp Schuster contributed to the evaluation, devising language plugins for bags and maps in collaboration with me.

1.1.2 Excluded papers

This thesis improves modularity of collection queries by automating different sorts of optimizations on them.

During my PhD work I collaborated on several other papers, mostly on different facets of modularity, which do not appear in this thesis.

Modularity Module boundaries hide information on implementation that is not relevant to clients. However, it often turns out that some scenarios, clients or task require such hidden information. As discussed, manual optimization requires hidden information: hence, researchers strive to automate optimizations. But other tasks often violate these boundaries. I collaborated on research on understanding why this happens and how to deal with this problem [Ostermann, Giarrusso, Kästner, and Rendel, 2011].

Software Product Lines In some scenarios, a different form of modular software is realized through software product lines (SPLs), where many *variants* of a single software artifact (with

different sets of features) can be generated. This is often done using conditional compilation, for instance through the C preprocessor.

But if a software product line uses conditional compilation, processing automatically its source code is difficult — in fact, even lexing or parsing it is a challenge. Since the number of variants is exponential in the number of features, generating and processing each variant does not scale.

To address these challenges, I collaborated with the TypeChef project to develop a variability-aware lexer [Kästner, Giarrusso, and Ostermann, 2011a] and parser [Kästner, Giarrusso, Rendel, Erdweg, Ostermann, and Berger, 2011b].

Another challenge is predicting non-functional properties (such as code size or performance) of a variant of a software product line before generation and direct measurement. Such predictions can be useful to select efficiently a variant that satisfies some non-functional requirements. I collaborated to research on predicting such non-functional properties: we measure the properties on a few variants and extrapolate the results to other variants. In particular, I collaborated to the experimental evaluation on a few open source projects, where even a simple model reached significant accuracy in a few scenarios [Siegmund, Rosenmüller, Kästner, Giarrusso, Apel, and Kolesnikov, 2011, 2013].

Domain-specific languages This thesis is concerned with DSLs, both ones for the domain of collection queries, but also (in Part II) more general ones.

Different forms of language composition, both among DSLs and across general-purpose languages and DSLs are possible, but their relationship is nontrivial; I collaborated to work classifying the different forms of language compositions [Erdweg, Giarrusso, and Rendel, 2012].

The implementation of SQuOpt, as described in Part I, requires an extensible representation of ASTs, similar to the one used by LMS [Rompf and Odersky, 2010]. While this representation is pretty flexible, it relies on Scala's support of GADTs [Emir et al., 2006, 2007b], which is known to be somewhat fragile due to implementation bugs. In fact, sound pattern matching on Scala's extensible GADTs is impossible without imposing significant restrictions to extensibility, due to language extensions not considered during formal study [Emir et al., 2006, 2007a]: the problem arises due to the interaction between GADTs, declaration-site variance annotations and variant refinement of type arguments at inheritance time. To illustrate the problem, I've shown it already applies to an extensible definitional interpreter for $\lambda_{<:}$ [Giarrusso, 2013]. While solutions have been investigated in other settings [Scherer and Rémy, 2013], the problem remains open for Scala to this day.

1.1.3 Navigating this thesis

The two parts of this thesis, while related by the common topic of collection queries, can be read mostly independently from each other. Summaries of the two parts are given respectively in Sec. 2.1 and Sec. 10.7.1.

Numbering We use numbering and hyperlinks to help navigation; we explain our conventions to enable exploiting them.

To simplify navigation, we number all sorts of "theorems" (including here definitions, remarks, even examples or descriptions of notation) per-chapter with counters including section numbers. For instance, Definition 12.2.1 is the first such item in Chapter 12, followed by Theorem 12.2.2, and they both appear in Sec. 12.2. And we can be sure that Lemma 12.2.8 comes after both "theorems" because of its number, even if they are of different sorts.

Similarly, we number tables and figures per-chapter with a shared counter. For instance, Fig. 6.1 is the first figure or table in Chapter 6, followed by Table 6.2.

To help reading, at times we will repeat or anticipate statements of "theorems" to refer to them, without forcing readers to jump back and forth. We will then reuse the original number. For instance Sec. 12.2 contains a copy of Slogan 10.3.3 with its original number.

For ease of navigation, all such references are hyperlinked in electronic versions of this thesis, and the table of contents is exposed to PDF readers via PDF bookmarks.

Part I

Optimizing Collection Queries by Reification

Chapter 2

Introduction

In-memory collections of data often need efficient processing. For on-disk data, efficient processing is already provided by database management systems (DBMS) thanks to their query optimizers, which support many optimizations specific to the domain of collections. Moving in-memory data to DBMSs, however, typically does not improve performance [Stonebraker et al., 2007], and query optimizers cannot be reused separately since DBMS are typically monolithic and their optimizers deeply integrated. A few collection-specific optimizations, such as shortcut fusion [Gill et al., 1993], are supported by compilers for purely functional languages such as Haskell. However, the implementation techniques for those optimizations do not generalize to many other ones, such as support for indexes. In general, collection-specific optimizations are not supported by the general-purpose optimizers used by typical (JIT) compilers.

Therefore programmers, when needing collection-related optimizations, perform them manually. To allow that, they are often forced to perform manual inlining [Peyton Jones and Marlow, 2002]. But manual inlining modifies source code by combining distinct functions together, while often distinct functions should remain distinct, because they deal with different concerns, or because one function need to be reused in a different context. In either case, manual inlining reduces modularity — defined here as the ability to abstract behavior in a separate function (possibly part of a different module) to enable reuse and improve understandability.

For these reasons, currently developers need to choose between modularity and performance, as also highlighted by Kiczales et al. [1997] on a similar example. Instead, we envision that they should rely on an automatic optimizer performing inlining and collection-specific optimizations. They would then achieve both performance and modularity.[1]

One way to implement such an optimizer would be to extend the compiler of the language with a collection-specific optimizer, or to add some kind of external preprocessor to the language. However, such solutions would be rather brittle (for instance, they lack composability with other language extensions) and they would preclude optimization opportunities that arise only at runtime.

For this reason, our approach is implemented as an embedded domain-specific language (EDSL), that is, as a regular library. We call this library SQuOpt, the Scala QUery OPTimizer. SQuOpt consists of a Scala EDSL for queries on collections based on the Scala collections API. An expression in this EDSL produces at run time an *expression tree* in the host language: a data structure which represents the query to execute, similar to an abstract syntax tree (AST) or a query plan. Thanks to

[1] In the terminology of Kiczales et al. [1997], our goal is to be able to decompose different *generalized procedures* of a program according to its primary decomposition, while separating the handling of some performance concerns. To this end, we are modularizing these performance concerns into a metaprogramming-based optimization module, which we believe could be called, in that terminology, *aspect*.

the extensibility of Scala, expressions in this language look almost identical to expressions with the same meaning in Scala. When executing the query, SQuOpt optimizes and compiles these expression trees for more efficient execution. Doing optimization at run time, instead of compile-time, avoids the need for control-flow analyses to determine which code will be actually executed [Chambers et al., 2010], as we will see later.

We have chosen Scala [Odersky et al., 2011] to implement our library for two reasons: (i) Scala is a good meta-language for EDSLs, because it is syntactically flexible and has a powerful type system, and (ii) Scala has a sophisticated collections library with an attractive syntax (for-comprehensions) to specify queries.

To evaluate SQuOpt, we study queries of the FindBugs tool [Hovemeyer and Pugh, 2004]. We rewrote a set of queries to use the Scala collections API and show that modularization incurs significant performance overhead. Subsequently, we consider versions of the same queries using SQuOpt. We demonstrate that the automatic optimization can reconcile modularity and performance in many cases. Adding advanced optimizations such as indexing can even improve the performance of the analyses beyond the original non-modular analyses.

2.1 Contributions and summary

Overall, our main contributions in Part I are the following:

- We illustrate the tradeoff between modularity and performance when manipulating collections, caused by the lack of domain-specific optimizations (Sec. 2.2). Conversely, we illustrate how domain-specific optimizations lead to more readable and more modular code (Chapter 3).
- We present the design and implementation of SQuOpt, an embedded DSL for queries on collections in Scala (Chapter 4).
- We evaluate SQuOpt to show that it supports writing queries that are at the same time modular and fast. We do so by re-implementing several code analyses of the FindBugs tool. The resulting code is more modular and/or more efficient, in some cases by orders of magnitude. In these case studies, we measured average speedups of 12x with a maximum of 12800x (Chapter 6).

2.2 Motivation

In this section, we show how the absense of collection-specific optimizations forces programmers to trade modularity against performance, which motivates our design of SQuOpt to resolve this conflict.

As our running example through the chapter, we consider representing and querying a simple in-memory bibliography. A book has, in our schema, a title, a publisher and a list of authors. Each author, in turn, has a first and last name. We represent authors and books as instances of the Scala classes `Author` and `Book` shown in Fig. 2.1. The class declarations list the type of each field: Titles, publishers, and first and last names are all stored in fields of type `String`. The list of authors is stored in a field of type `Seq[Author]`, that is, a sequence of authors – something that would be more complex to model in a relational database. The code fragment also defines a collection of books named `books`.

As a common idiom to query such collections, Scala provides *for-comprehensions*. For instance, the for-comprehension computing `records` in Fig. 2.2 finds all books published by Pearson Education and yields, for each of those books, and for each of its authors, a record containing the book title,

```
package schema
case class Author(firstName: String, lastName: String)
case class Book(title: String, publisher: String,
  authors: Seq[Author])

val books: Set[Book] = Set(
  new Book("Compilers: Principles, Techniques and Tools",
       "Pearson Education",
       Seq(new Author("Alfred V.", "Aho"),
           new Author("Monica S.", "Lam"),
           new Author("Ravi", "Sethi"),
           new Author("Jeffrey D.", "Ullman"))
  /* other books... */)
```

Figure 2.1: Definition of the schema and of some content.

the full name of that author and the number of additional coauthors. The *generator* `book` ← `books` functions like a loop header: The remainder of the for-comprehension is executed once per book in the collection. Consequently, the *generator* `author` ← `book.authors` starts a nested loop. The return value of the for-comprehension is a collection of all yielded records. Note that if a book has multiple authors, this for-comprehensions will return multiple records relative to this book, one for each author.

We can further process this collection with another for-comprehension, possibly in a different module. For example, still in Fig. 2.2, the function `titleFilter` filters book titles containing the word "Principles", and drops from each record the number of additional coauthors.

In Scala, the implementation of for-comprehensions is not fixed. Instead, the compiler desugars a for-comprehension to a series of API calls, and different collection classes can implement this API differently. Later, we will use this flexibility to provide an optimizing implementation of for-comprehensions, but in this section, we focus on the behavior of the standard Scala collections, which implement for-comprehensions as loops that create intermediate collections.

2.2.1 Optimizing by Hand

In the naive implementation in Fig. 2.2 different concerns are separated, hence it is modular. However, it is also inefficient. To execute this code, we first build the original collection and only later we perform further processing to build the new result; creating the intermediate collection at the interface between these functions is costly. Moreover, the same book can appear in `records` more than once if the book has more than one author, but all of these duplicates have the same title. Nevertheless, we test each duplicate title separately whether it contains the searched `keyword`. If books have 4 authors on average, this means a slowdown of a factor of 4 for the filtering step.

In general, one can only resolve these inefficiencies by manually optimizing the query; however, we will observe that these manual optimizations produce less modular code.[2]

To address the first problem above, that is, to avoid creating intermediate collections, we can manually inline `titleFilter` and `records`; we obtain two nested for-comprehensions. Furthermore, we can *unnest* the inner one [Fegaras and Maier, 2000].

To address the second problem above, that is, to avoid testing the same title multiple times, we *hoist* the filtering step, that is, we change the order of the processing steps in the query to first

[2]The existing Scala collections API supports optimization, for instance through non-strict variants of the query operators (called 'views' in Scala), but they can only be used for a limited set of optimizations, as we discuss in Chapter 8.

```
case class BookData(title: String, authorName: String,
  coauthors: Int)

val records =
  for {
    book ← books
    if book.publisher == "Pearson Education"
    author ← book.authors
  } yield new BookData(book.title,
                 author.firstName + " " +
                 author.lastName,
                 book.authors.size - 1)

def titleFilter(records: Set[BookData],
    keyword: String) =
  for {
    record ← records
    if record.title.contains(keyword)
  } yield (record.title, record.authorName)

val res = titleFilter(records, "Principles")
```

Figure 2.2: Our example query on the schema in Fig. 2.1, and a function which postprocesses its result.

look for `keyword` within `book.title` and then iterate over the set of authors. This does not change the overall semantics of the query because the filter only accesses the title but does not depend on the author. In the end, we obtain the code in Fig. 2.3. The resulting query processes the title of each book only once. Since filtering in Scala is done lazily, the resulting query avoids building an intermediate collection.

This second optimization is only possible after inlining and thereby reducing the modularity of the code, because it mixes together processing steps from `titleFilter` and from the definition of `records`. Therefore, reusing the code creating records would now be harder.

To make `titleFilterHandOpt` more reusable, we could turn the publisher name into a parameter. However, the new versions of `titleFilter` cannot be reused as-is if some details of the inlined code change; for instance, we might need to filter publishers differently or not at all. On the other hand, if we express queries modularly, we might lose some opportunities for optimization. The design of the collections API, both in Scala and in typical languages, forces us to manually optimize our code by repeated inlining and subsequent application of query optimization rules, which leads to a loss of modularity.

```
def titleFilterHandOpt(books: Set[Book],
                       publisher: String,
                       keyword: String) =
  for {
    book ← books
    if book.publisher == publisher && book.title.contains(keyword)
    author ← book.authors
  } yield (book.title, author.firstName + " " + author.lastName)
val res = titleFilterHandOpt(books,
  "Pearson Education", "Principles")
```

Figure 2.3: Composition of queries in Fig. 2.2, after inlining, query unnesting and hoisting.

Chapter 3

Automatic optimization with SQuOpt

The goal of SQuOpt is to let programmers write queries modularly and at a high level of abstraction and deal with optimization by a dedicated domain-specific optimizer. In our concrete example, programmers should be able to write queries similar to the one in Fig. 2.2, but get the efficiency of the one in Fig. 2.3. To allow this, SQuOpt overloads for-comprehensions and other constructs, such as string concatenation with + and field access `book.author`. Our overloads of these constructs reify the query as an expression tree. SQuOpt can then optimize this expression tree and execute the resulting optimized query. Programmers explicitly trigger processing by SQuOpt, by adapting their queries as we describe in next subsection.

3.1 Adapting a Query

To use SQuOpt instead of native Scala queries, we first assume that the query does not use side effects and is thus *purely functional*. We argue that purely functional queries are more declarative. Side effects are used to improve performance, but SQuOpt makes that unnecessary through automatic optimizations. In fact, the lack of side effects enables more optimizations.

In Fig. 3.1 we show a version of our running example adapted to use SQuOpt. We first discuss changes to `records`. To enable SQuOpt, a programmer needs to (a) import the SQuOpt library, (b) import some wrapper code specific to the types the collection operates on, in this case `Book` and `Author` (more about that later), (c) convert explicitly the native Scala collections involved to collections of our framework by a call to `asSquopt`, (d) rename a few operators such as == to ==# (this is necessary due to some Scala limitations), and (e) add a separate step where the query is evaluated (possibly after optimization). All these changes are lightweight and mostly of a syntactic nature.

For parameterized queries like `titleFilter`, we need to also adapt type annotations. The ones in `titleFilterQuery` reveal some details of our implementation: Expressions that are reified have type `Exp[T]` instead of `T`. As the code shows, `resQuery` is optimized before compilation. This call will perform the optimizations that we previously did by hand and will return a query equivalent to that in Fig. 2.3, after verifying their safety conditions. For instance, after inlining, the filter `if book.title.contains(keyword)` does not reference `author`; hence, it is safe to hoist. Note that checking this safety condition would not be possible without reifying the predicate. For instance, it would not be sufficient to only reify the calls to the collection API, because the predicate is represented as a boolean function parameter. In general, our automatic optimizer inspects the whole

```
import squopt._
import schema.squopt._

val recordsQuery =
  for {
    book ← books.asSquopt
    if book.publisher ==# "Pearson Education"
    author ← book.authors
  } yield new BookData(book.title,
    author.firstName + " " + author.lastName,
    book.authors.size - 1)

//...
val records = recordsQuery.eval

def titleFilterQuery(records: Exp[Set[BookData]], keyword: Exp[String]) =
  for {
    record ← records
    if record.title.contains(keyword)
  } yield (record.title, record.authorName)

val resQuery = titleFilterQuery(recordsQuery, "Principles")
val res = resQuery.optimize.eval
```

Figure 3.1: SQuOpt version of Fig. 2.2; `recordsQuery` contains a reification of the query, `records` its result.

reification of the query implementation to check that optimizations do not introduce changes in the overall result of the query and are therefore safe.

3.2 Indexing

SQuOpt also supports the transparent usage of indexes. Indexes can further improve the efficiency of queries, sometimes by orders of magnitude. In our running example, the query scans all books to look for the ones having the right publisher. To speed up this query, we can preprocess `books` to build an index, that is, a dictionary mapping, from each publisher to a collection of all the books it published. This index can then be used to answer the original query without scanning all books.

We construct a *query* representing the desired dictionary, and inform the optimizer that it should use this index where appropriate:

```
val idxByPublisher = books.asSquopt.indexBy(_.publisher)
Optimization.addIndex(idxByPublisher)
```

The `indexBy` collection method accepts a function that maps a collection element to a key; `coll.indexBy(key)` returns a dictionary mapping each key to the collection of all elements of `coll` having that key. Missing keys are mapped to an empty collection.[1] `Optimization.addIndex` simply preevaluates the index and updates a dictionary mapping the index to its preevaluated result.

[1]For readers familiar with the Scala collection API, we remark that the only difference with the standard `groupBy` method is the handling of missing keys.

A call to `optimize` on a query will then take this index into account and rewrite the query to perform index lookup instead of scanning, if possible. For instance, the code in Fig. 3.1 would be transparently rewritten by the optimizer to a query similar to the following:

```
val indexedQuery =
  for {
    book ← idxByPublisher("Pearson Education")
    author ← book.authors
  } yield new BookData(book.title,
    author.firstName + " " + author.lastName,
    book.authors.size - 1)
```

Since dictionaries in Scala are functions, in the above code, dictionary lookup on `idxByPublisher` is represented simply as function application. The above code iterates over books having the desired publisher, instead of scanning the whole library, and performs the remaining computation from the original query. Although the index use in the listing above is written as `idxByPublisher("Pearson Education")`, only the cached result of evaluating the index is used when the query is executed, not the reified index definition.

This optimization could also be performed manually, of course, but the queries are on a higher abstraction level and more maintainable if indexing is defined separately and applied automatically. Manual application of indexing is a crosscutting concern because adding or removing an index affects potentially many queries. SQuOpt does not free the developer from the task of assessing which index will 'pay off' (we have not considered automatic index creation yet), but at least it becomes simple to add or remove an index, since the application of the indexes is modularized in the optimizer.

Chapter 4

Implementing SQuOpt

After describing how to use SQuOpt, we explain how SQuOpt represents queries internally and optimizes them. Here we give only a brief overview of our implementation technique; it is described in more detail in Sec. 5.1.

4.1 Expression Trees

In order to analyze and optimize collection queries at runtime, SQuOpt reifies their syntactic structure as *expression trees*. The expression tree reflects the syntax of the query after desugaring, that is, after for-comprehensions have been replaced by API calls. For instance, `recordsQuery` from Fig. 3.1 points to the following expression tree (with some boilerplate omitted for clarity):

```
new FlatMap(
  new Filter(
    new Const(books),
    v2 ⇒ new Eq(new Book_publisher(v2),
                new Const("Pearson Education"))),
    v3 ⇒ new MapNode(
          new Book_authors(v3),
          v4 ⇒ new BookData(
                 new Book_title(v3),
                 new StringConcat(
                   new StringConcat(
                     new Author_firstName(v4),
                     new Const(" ")),
                   new Author_lastName(v4)),
                 new Plus(new Size(new Book_authors(v3)),
                          new Negate(new Const(1))))))
```

The structure of the for-comprehension is encoded with the `FlatMap`, `Filter` and `MapNode` instances. These classes correspond to the API methods that for-comprehensions get desugared to. SQuOpt arranges for the implementation of `flatMap` to construct a `FlatMap` instance, etc. The instances of the other classes encode the rest of the structure of the collection query, that is, which methods are called on which arguments. On the one hand, SQuOpt defines classes such as `Const` or `Eq` that are generic and applicable to all queries. On the other hand, classes such as `Book_publisher` cannot be predefined, because they are specific to the user-defined types used in a query. SQuOpt provides a small code generator, which creates a case class for each method and field of a user-

defined type. Functions in the query are represented by functions that create expression trees; representing functions in this way is frequently called higher-order abstract syntax [Pfenning and Elliot, 1988].

We can see that the reification of this code corresponds closely to an abstract syntax tree for the code which is executed; however, many calls to specific methods, like `map`, are represented by special nodes, like `MapNode`, rather than as method calls. For the optimizer it becomes easier to match and transform those nodes than with a generic abstract syntax tree.

Nodes for collection operations are carefully defined by hand to provide them highly generic type signatures and make them reusable for all collection types. In Scala, collection operations are highly polymorphic; for instance, `map` has a single implementation working on all collection types, like `List`, `Set`, and we similarly want to represent all usages of `map` through instances of a single node type, namely `MapNode`. Having separate nodes `ListMapNode`, `SetMapNode` and so on would be inconvenient, for instance when writing the optimizer. However, `map` on a `List[Int]` will produce another `List`, while on a `Set` it will produce another `Set`, and so on for each specific collection type (in first approximation); moreover, this is guaranteed statically by the type of `map`. Yet, thanks to advanced typesystem features, `map` is defined only once avoiding redundancy, but has a type polymorphic enough to guarantee statically that the correct return value is produced. Since our tree representation is strongly typed, we need to have a similar level of polymorphism in `MapNode`. We achieved this by extending the techniques described by Odersky and Moors [2009], as detailed in our technical report [Giarrusso et al., 2012].

We get these expression trees by using Scala implicit conversions in a particular style, which we adopted from Rompf and Odersky [2010]. Implicit conversions allow to add, for each method `A.foo(B)`, an overload of `Exp[A].foo(Exp[B])`. Where a value of type `Exp[T]` is expected, a value of type `T` can be used thanks to other implicit conversions, which wrap it in a `Const` node. The initial call of `asSquopt` triggers the application of the implicit conversions by converting the collection to the leaf of an expression tree.

It is also possible to call methods that do not return expression trees; however, such method calls would then only be represented by an opaque `MethodCall` node in the expression tree, which means that the code of the method cannot be considered in optimizations.

Crucially, these expression trees are generated at runtime. For instance, the first `Const` contains a reference to the actual collection of books to which `books` refers. If a query uses another query, such as `records` in Fig. 3.1, then the subquery is effectively *inlined*. The same holds for method calls inside queries: If these methods return an expression tree (such as the `titleFilterQuery` method in Fig. 3.1), then these expression trees are inlined into the composite query. Since the reification happens at runtime, it is not necessary to predict the targets of dynamically bound method calls: A new (and possibly different) expression tree is created each time a block of code containing queries is executed.

Hence, we can say that expression trees represent the computation which is going to be executed after inlining; control flow or virtual calls in the original code typically disappear — especially if they manipulate the query as a whole. This is typical of deeply embedded DSLs like ours, where code instead of performing computations produces a representation of the computation to perform [Elliott et al., 2003; Chambers et al., 2010].

This inlining can duplicate computations; for instance, in this code:

```
val num: Exp[Int] = 10
val square = num * num
val sum = square + square
```

evaluating `sum` will evaluate `square` twice. Elliott et al. [2003] and we avoid this using common-subexpression elimination.

4.2 Optimizations

Our optimizer currently supports several algebraic optimizations. Any query and in fact every reified expression can be optimized by calling the `optimize` function on it. The ability to optimize reified expressions that are not queries is useful; for instance, optimizing a function that produces a query is similar to a "prepared statement" in relational databases.

The optimizations we implemented are mostly standard in compilers [Muchnick, 1997] or databases:

- *Query unnesting* merges a nested query into the containing one [Fegaras and Maier, 2000; Grust and Scholl, 1999], replacing for instance

  ```
  for {val1 ← (for {val2 ← coll} yield f(val2))}
    yield g(val1)
  ```

  with

  ```
  for {val2 ← coll; val1 = f(val2)} yield g(val1)
  ```

- *Bulk operation fusion* fuses higher-order operators on collections.
- *Filter hoisting* tries to apply filters as early as possible; in database query optimization, it is known as selection pushdown. For filter hoisting, it is important that the full query is reified, because otherwise the dependencies of the filter condition cannot be determined.
- We reduce during optimization tuple/case class accesses: For instance, `(a, b)._1` is simplified to `a`. This is important because the produced expression does not depend on `b`; removing this false dependency can allow, for instance, a filter containing this expression to be hoisted to a context where `b` is not bound.
- *Indexing* tries to apply one or more of the available indexes to speed up the query.
- *Common subexpression elimination (CSE)* avoids that the same computation is performed multiple times; we use techniques similar to Rompf and Odersky [2010].
- Smaller optimizations include constant folding, reassociation of associative operators and removal of identity maps (`coll.map(x ⇒ x)`, typically generated by the translation of for-comprehensions).

Each optimization is applied recursively bottom-up until it does not trigger anymore; different optimizations are composed in a fixed pipeline.

Optimizations are only guaranteed to be semantics-preserving if queries obey the restrictions we mentioned: for instance, queries should not involve side-effects such as assignments or I/O, and all collections used in queries should implement the specifications stated in the collections API. Obviously the choice of optimizations involves many tradeoffs; for that reason we believe that it is all the more important that the optimizer is not hard-wired into the compiler but implemented as a library, with potentially many different implementations.

To make changes to the optimizer more practical, we designed our query representation so that optimizations are easy to express; restricting to pure queries also helps. For instance, filter fusion can be implemented in few lines of code: just match against expressions of form `coll.filter(pred2).filter(pred1)` and rewrite them:[1]

[1] `Sym` nodes are part of the boilerplate we omitted earlier.

```
val mergeFilters = ExpTransformer {
  case Sym(Filter(Sym(Filter(collection, pred2)), pred1)) ⇒
    coll.filter(x ⇒ pred2(x) && pred1(x))
}
```

A more complex optimization such as filter hoisting requires only 20 lines of code.

We have implemented a prototype of the optimizer with the mentioned optimizations. Many additional algebraic optimizations can be added in future work by us or others; a candidate would be loop hoisting, which moves out of loops arbitrary computations not depending on the loop variable (and not just filters). With some changes to the optimizer's architecture, it would also be possible to perform cost-based and dynamic optimizations.

4.3 Query Execution

Calling the `eval` method on a query will convert it to executable bytecode; this bytecode will be loaded and invoked by using Java reflection. We produce a thunk that, when evaluated, will execute the generated code.

In our prototype we produce bytecode by converting expression trees to Scala code and invoking on the result the Scala compiler, `scalac`. Invoking `scalac` is typically quite slow, and we currently use caching to limit this concern; however, we believe it is merely an engineering problem to produce bytecode directly from expression trees, just as compilers do.

Our expression trees contain native Scala values wrapped in `Const` nodes, and in many cases one cannot produce Scala program text evaluating to the same value. To allow executing such expression trees we need to implement cross-stage persistence (CSP): the generated code will be a function, accepting the actual values as arguments [Rompf and Odersky, 2010]. This allows sharing the compiled code for expressions which differ only in the embedded values.

More in detail, our compilation algorithm is as follows. (a) We implement CSP by replacing embedded Scala values by references to the function arguments; so for instance `List(1, 2, 3).map(x ⇒ x + 1)` becomes the function `(s1: List[Int], s2: Int) ⇒ s1.map(x ⇒ x + s2)`. (b) We look up the produced expression tree, together with the types of the constants we just removed, in a cache mapping to the generated classes. If the lookup fails we update the cache with the result of the next steps. (c) We apply CSE on the expression. (d) We convert the tree to code, compile it and load the generated code.

Preventing errors in generated code Compiler errors in generated code are typically a concern; with SQuOpt, however, they can only arise due to implementation bugs in SQuOpt (for instance in pretty-printing, which cannot be checked statically), so they do not concern users. Since our query language and tree representation are statically typed, type-incorrect queries will be rejected statically. For instance, consider again `idxByPublisher`, described previously:

```
val idxByPublisher = books.asSquopt.indexBy(_.publisher)
```

Since `Book.publisher` returns a `String`, `idxByPublisher` has type `Exp[Map[String, Book]]`. Looking up a key of the wrong type, for instance by writing `idxByPublisher(book)` where `book: Book`, will make `scalac` emit a static type error.

Chapter 5

A deep EDSL for collection queries

In this chapter we discuss collections as a *critical* case study [Flyvbjerg, 2006] for deep DSL embedding.

As discussed in the previous chapter, to support optimizations we require a deep embedding of the collections DSL.

While the basic idea of deep embedding is well known, it is not obvious how to realize deep embedding when considering the following additional goals:

- To support users adopting SQuOpt, a generic SQuOpt query should share the "look and feel" of the ordinary collections API: In particular, query syntax should remain mostly unchanged. In our case, we want to preserve Scala's *for-comprehension*[1] syntax and its notation for anonymous functions.

- Again to support users adopting SQuOpt, a generic SQuOpt query should not only share the syntax of the ordinary collections API; it should also be well-typed if and only if the corresponding ordinary query is well-typed. This is particularly challenging in the Scala collections library due to its deep integration with advanced type-system features, such as higher-kinded generics and implicit objects [Odersky and Moors, 2009]. For instance, calling `map` on a `List` will return a `List`, and calling `map` on a `Set` will return a `Set`. Hence the object-language representation and the transformations thereof should be as "typed" as possible. This precludes, among others, a first-order representation of object-language variables as strings.

- SQuOpt should be interoperable with ordinary Scala code and Scala collections. For instance, it should be possible to call normal non-reified functions within a SQuOpt query, or mix native Scala queries and SQuOpt queries.

- The performance of SQuOpt queries should be reasonable even without optimizations. A non-optimized SQuOpt query should not be dramatically slower than a native Scala query. Furthermore, it should be possible to create new queries at run time and execute them without excessive overhead. This goal limits the options of applicable interpretation or compilation techniques.

We think that these challenges characterize deep embedding of queries on collections as a *critical* case study [Flyvbjerg, 2006] for DSL embedding. That is, it is so challenging that embedding

[1] Also known as *for expressions* [Odersky et al., 2011, Ch. 23].

techniques successfully used in this case are likely to be successful on a broad range of other DSLs. In this chapter we report, from the case study, the successes and failures of achieving these goals in SQuOpt.

5.1 Implementation: Expressing the interface in Scala

To optimize a query as described in the previous section, SQuOpt needs to reify, optimize and execute queries. Our implementation assigns responsibility for these steps to three main components: A generic library for reification and execution of general Scala expressions, a more specialized library for reification and execution of query operators, and a dedicated query optimizer. Queries need then to be executed through either compilation (already discussed in Sec. 4.3) or interpretation (to discuss in Sec. 5.6). We describe the implementation in more detail in the rest of this section. The full implementation is also available online[2].

A core idea of SQuOpt is to reify Scala code as a data structure in memory. A programmer could directly create instances of that data structure, but we also provide a more convenient interface based on advanced Scala features such as implicit conversions and type inference. That interface allows to automatically reify code with a minimum of programmer annotations, as shown in the examples in Chapter 3. Since this is a case study on Scala's support for deep embedding of DSLs, we also describe in this section how Scala supports our task. In particular, we report on techniques we used and issues we faced.

5.2 Representing expression trees

In the previous section, we have seen that expressions that would have type `T` in a native Scala query are reified and have type `Exp[T]` in SQuOpt. The generic type `Exp[T]` is the base for our reification of Scala expression as expression trees, that is, as data structures in memory. We provide a subclass of `Exp[T]` for every different form of expression we want to reify. For example, in Fig. 2.2 the expression `author.firstName + " " + author.lastName` must be reified even though it is not collection-related, for otherwise the optimizer could not see whether `author` is used. Knowing this is needed for instance to remove variables which are bound but not used. Hence, this expression is reified as

```
StringConcat(StringConcat(AuthorFirstName(author), Const(" ")),
  AuthorLastName(author))
```

This example uses the constructors of the following subclasses of `Exp[T]` to create the expression tree.

```
case class Const[T](t: T) extends Exp[T]
case class StringConcat(str1: Exp[String], str2: Exp[String])
  extends Exp[String]
case class AuthorFirstName(t: Exp[Author]) extends Exp[String]
case class AuthorLastName(t: Exp[Author]) extends Exp[String]
```

Expression nodes additionally implement support code for tree traversals to support optimizations, which we omit here.

This representation of expression trees is well-suited for a representation of the structure of expressions in memory and also for pattern matching (which is automatically supported for case classes in Scala), but inconvenient for query writers. In fact, in Fig. 3.1, we have seen that SQuOpt

[2] http://www.informatik.uni-marburg.de/~pgiarrusso/SQuOpt

provides a much more convenient front-end: The programmer writes almost the usual code for type `T` and SQuOpt converts it automatically to `Exp[T]`.

5.3 Lifting first-order expressions

We call the process of converting from `T` to `Exp[T]` *lifting*. Here we describe how we lift first-order expressions – Scala expressions that do not contain anonymous function definitions.

To this end, consider again the fragment

```
author.firstName + " " + author.lastName
```

now in the context of the SQuOpt-enabled query in Fig. 3.1. It looks like a normal Scala expression, even syntactically unchanged from Fig. 2.2. However, evaluating that code in the context of Fig. 3.1 does not concatenate any strings, but creates an expression tree instead. Although the code looks like the same expression, it has a different *type*, `Exp[String]` instead of `String`. This difference in the type is caused by the context: The variable `author` is now bound in a SQuOpt-enabled query and therefore has type `Exp[Author]` instead of `Author`. We can still access the `firstName` field of `author`, because expression trees of type `Exp[T]` provide the same interface as values of type `T`, except that all operations return expressions trees instead of values.

To understand how an expression tree of type `Exp[T]` can have the same interface as a value of type `T`, we consider two expression trees `str1` and `str2` of type `Exp[String]`. The implementation of lifting differs depending on the kind of expression we want to lift.

Method calls and operators In our example, operator `+` should be available on `Exp[String]`, but not on `Exp[Boolean]`, because `+` is available on `String` but not on `Boolean`. Furthermore, we want `str1 + str2` to have type `Exp[String]` and to evaluate not to a string concatenation but to a call of `StringConcat`, that is, to an expression tree which *represents* `str1 + str2`. This is a somewhat unusual requirement, because usually, the interface of a generic type does not depend on the type parameters.

To provide such operators and to encode expression trees, we use *implicit conversions* in a similar style as Rompf and Odersky [2010]. Scala allows to make expressions of a type `T` implicitly convertible to a different type `U`. To this end, one must declare an *implicit conversion function* having type `T ⇒ U`. Calls to such functions will be inserted by the compiler when required to fix a type mismatch between an expression of type `T` and a context expecting a value of type `U`. In addition, a method call `e.m(args)` can trigger the conversion of `e` to a type where the method `m` is present[3]. Similarly, an operator usage, as in `str1 + str2`, can also trigger an implicit conversion: an expression using an operator, like `str1 + str2`, is desugared to the method call `str1.+(str2)`, which can trigger an implicit conversion from `str1` to a type providing the `+` method. Therefore from now on we do not distinguish between operators and methods.

To provide the method `+` on `Exp[String]`, we define an implicit conversion from `Exp[String]` to a new type providing a `+` method which creates the appropriate expression node.

```
implicit def expToStringOps(t: Exp[String]) = new StringOps(t)
class StringOps(t: Exp[String]) {
  def +(that: Exp[String]): Exp[String] = StringConcat(t, that)
}
```

This is an example of the well-known Scala *enrich-my-library pattern*[4] [Odersky, 2006].

With these declarations in scope, the Scala compiler rewrites `str1 + str2` to `expToStringOps(str1).+(str2)`, which evaluates to `StringConcat(str1, str2)` as desired. The implicit conversion

[3] For the exact rules, see Odersky et al. [2011, Ch. 21] and Odersky [2011].

[4] Also known as *pimp-my-library* pattern.

function `expToStringOps` is not applicable to `Exp[Boolean]` because it explicitly specifies the receiver of the +-call to have type `Exp[String]`. In other words, expressions like `str1 + str2` are now *lifted* on the level of expression trees in a type-safe way. For brevity, we refer to the defined operator as `Exp[String].+`.

Literal values However, string concatenation might also be applied to constants, as in `str1 + "foo"` or `"bar" + str1`. To lift `str1 + "foo"`, we introduce a lifting for constants which wraps them in a `Const` node:

```
implicit def pure[T](t: T): Exp[T] = Const(t)
```

The compiler will now rewrite `str1 + "foo"` to `expToStringOps(str1) + pure("foo")`, and `str1 + "foo" + str2` to `expToStringOps(str1) + pure("foo") + str2`. Different implicit conversions cooperate successfully to lift the expression.

Analogously, it would be convenient if the similar expression `"bar" + str1` would be rewritten to `expToStringOps(pure("bar")) + str1`, but this is not the case, because implicit coercions are not chained automatically in Scala. Instead, we have to manually chain existing implicit conversions into a new one:

```
implicit def toStringOps(t: String) = expToStringOps(pure(t))
```

so that `"bar" + str1` is rewritten to `toStringOps("bar") + str1`.

User-defined methods Calls of user-defined methods like `author.firstName` are lifted the same way as calls to built-in methods such as string concatenation shown earlier. For the running example, the following definitions are necessary to lift the methods from `Author` to `Exp[Author]`.

```
package schema.squopt

implicit def expToAuthorOps(t: Exp[Author]) = new AuthorOps(t)
implicit def toAuthorOps(t: Author) = expToAuthorOps(pure(t))

class AuthorOps(t: Exp[Author]) {
  def firstName: Exp[String] = AuthorFirstName(t)
  def lastName: Exp[String] = AuthorLastName(t)
}
```

Implicit conversions for user-defined types cooperate with other ones; for instance, `author.firstName + " " + author.lastName` is rewritten to

```
(expToStringOps(expToAuthorOps(author).firstName) + pure(" ")) +
  expToAuthorOps(author).lastName
```

`Author` is not part of SQuOpt or the standard Scala library, but an application-specific class, hence SQuOpt cannot pre-define the necessary lifting code. Instead, the application programmer needs to provide this code to connect SQuOpt to his application. To support the application programmer with this tedious task, we provide a code generator which discovers the interface of a class through reflection on its compiled version and generates the boilerplate code such as the one above for `Author`. It also generates the application-specific expression tree types such as `AuthorFirstName` as shown in Sec. 5.2. In general, query writers need to generate and import the boilerplate lifting code for all application-specific types they want to use in a SQuOpt query.

If desired, we can exclude some methods to *restrict* the language supported in our deeply embedded programs. For instance, SQuOpt requires the user to write side-effect-free queries, hence we do not lift methods which perform side effects.

Using similar techniques, we can also lift existing functions and implicit conversions.

Tuples and other generic constructors The techniques presented above for the lifting of method calls rely on overloading the name of the method with a signature that involves `Exp`. Implicit

resolution (for method calls) will then choose our lifted version of the function or method to satisfy the typing requirements of the context or arguments of the call. Unfortunately, this technique does not work for tuple constructors, which, in Scala, are not resolved like ordinary calls. Instead, support for tuple types is hard-wired into the language, and tuples are always created by the predefined tuple constructors.

For example, the expression `(str1, str2)` will always call Scala's built-in `Tuple2` constructor and correspondingly have type `(Exp[String], Exp[String])`. We would prefer that it calls a lifting function and produces an expression tree of type `Exp[(String, String)]` instead.

Even though we cannot intercept the call to `Tuple2`, we can add an implicit conversion to be called *after* the tuple is constructed.

```
implicit def tuple2ToTuple2Exp[A1, A2](tuple: (Exp[A1], Exp[A2])):
    LiftTuple2[A1, A2] = LiftTuple2[A1, A2](tuple._1, tuple._2)
case class LiftTuple2[A1, A2](t1: Exp[A1], t2: Exp[A2])
  extends Exp[(A1, A2)]
```

We generate such conversions for different arities with a code generator. These conversions will be used only when the context requires an expression tree. Note that this technique is only applicable because tuples are generic and support arbitrary components, including expression trees.

In fact, we use the same technique also for other generic constructors to avoid problems associated with shadowing of constructor functions. For example, an implicit conversion is used to lift `Seq[Exp[T]]` to `Exp[Seq[T]]`: code like `Seq(str1, str2)` first constructs a sequence of expression trees and then wraps the result with an expression node that describes a sequence.

Subtyping So far, we have seen that for each first-order method `m` operating on instances of `T`, we can create a corresponding method which operates on `Exp[T]`. If the method accepts parameters having types `A1, ... , An` and has return type `R`, the corresponding lifted method will accept parameters having types `Exp[A1], ... , Exp[An]` and return type `Exp[R]`. However, `Exp[T]` also needs to support all methods that `T` inherits from its super-type `S`. To ensure this, we declare the type constructor `Exp` to be *covariant* in its type parameter, so that `Exp[T]` correctly inherits the liftings from `Exp[S]`. This works even with the enrich-my-library pattern because implicit resolution respects subtyping in an appropriate way.

Limitations of Lifting Lifting methods of `Any` or `AnyRef` (Scala types at the root of the inheritance hierarchy) is not possible with this technique: `Exp[T]` inherits such methods and makes them directly available, hence the compiler will not insert an implicit conversion. Therefore, it is not possible to lift expressions such as `x == y`; rather, we have to rely on developer discipline to use `==\#` and `!=\#` instead of == and !=.

An expression like `"foo" + "bar" + str1` is converted to

```
toStringOps("foo" + "bar") + str1
```

Hence, part of the expression is evaluated before being reified. This is harmless here since we want `"foo" + "bar"` to be evaluated at compile-time, that is constant-folded, but in other cases it is preferable to prevent the constant folding. We will see later examples where queries on collections are evaluated before reification, defeating the purpose of our framework, and how we work around those.

5.4 Lifting higher-order expressions

We have shown how to lift first-order expressions; however, the interface of collections also uses higher-order methods, that is, methods that accept functions as parameters, and we need to lift them as well to reify the complete collection EDSL. For instance, the `map` method applies a function to

each element of a collection. In this section, we describe how we reify such expressions of function type.

Higher-order abstract syntax To represent functions, we have to represent the bound variables in the function bodies. For example, a reification of `str ⇒ str + "!"` needs to reify the variable `str` of type `String` in the body of the anonymous function. This reification should retain the information where `str` is bound. We achieve this by representing bound variables using *higher-order abstract syntax* (HOAS) [Pfenning and Elliot, 1988], that is, we represent them by variables bound at the meta level. To continue the example, the above function is reified as `(str: Exp[String]) ⇒ str + "!"`. Note how the type of `str` in the body of this version is `Exp[String]`, because `str` is a reified variable now. Correspondingly, the expression `str + "!"` is lifted as described in the previous section.

With all operations in the function body automatically lifted, the only remaining syntactic difference between normal and lifted functions is the type annotation for the function's parameter. Fortunately, Scala's *local type inference* can usually deduce the argument type from the context, for example, from the signature of the `map` operation being called. Type inference plays a dual role here: First, it allows the query writer to leave out the annotation, and second, it triggers lifting in the function body by requesting a lifted function instead of a normal function. This is how in Fig. 3.1, a single call to `asSquopt` triggers lifting of the overall query.

Reified functions have type `Exp[A] ⇒ Exp[B]` instead of the more regular `Exp[A ⇒ B]`. We chose the former over the latter to support Scala's syntax for anonymous functions and for-comprehensions which is hard-coded to produce or consume instances of the pre-defined `A ⇒ B` type. We have to reflect this irregularity in the lifting of methods and functions by treating the types of higher-order arguments accordingly.

User-defined methods, revised We can now extend the lifting of signatures for methods or functions from the previous section to the general case, that is, the case of higher-order functions. We lift a method or function with signature

```
def m[A1, ..., An](a1: T1, ..., an: Tn): R
```

to a method or function with the following signature.

```
def m[A1, ..., An](a1: Lift⟦T1⟧, ..., an: Lift⟦Tn⟧): Lift⟦R⟧
```

As before, the definition of the lifted method or function will return an expression node representing the call. If the original was a function, the lifted version is also defined as a function. If the original was a method on type `T`, then the lifted version is enriched onto `T`.

The type transformation *Lift* converts the argument and return types of the method or function to be lifted. For most types, *Lift* just wraps the type in the `Exp` type constructor, but function types are treated specially: *Lift* recursively descends into function types to convert their arguments separately. Overall, *Lift* behaves as follows.

```
Lift⟦(A1, . . . , An) ⇒ R]⟧ =
  (Lift⟦A1⟧, ... , Lift⟦An⟧ ⇒ Lift⟦R⟧
Lift⟦A⟧ = Exp[A]
```

We can use this extended definition of method lifting to implement `map` lifting for `Lists`, that is, a method with signature `Exp[List[T]].map(Exp[T] ⇒ Exp[U])`:

```
implicit def expToListOps[T](coll: Exp[List[T]]) = new ListOps(coll)
implicit def toListOps[T](coll: List[T]) = expToListOps(coll)

class ListOps(coll: Exp[List[T]]) {
  def map[U](f: Exp[T] ⇒ Exp[U]) = ListMapNode(coll, Fun(f))
}
```

```
val records = books.
  withFilter(book ⇒ book.publisher == "Pearson Education").
  flatMap(book ⇒ book.authors.
  map(author ⇒ BookData(book.title,
    author.firstName + " " + author.lastName,
    book.authors.size - 1)))
```

Figure 5.1: Desugaring of code in Fig. 2.2.

```
case class ListMapNode[T, U](coll: Exp[List[T]], mapping: Exp[T ⇒ U])
  extends Exp[List[U]]
```

Note how `map`'s parameter `f` has type `Exp[T] ⇒ Exp[U]` as necessary to enable Scala's syntax for anonymous functions and automatic lifting of the function body. This implementation would work for queries on lists, but does not support other collection types or queries where the collection type changes. We show in Sec. 5.5 how SQuOpt integrates with such advanced features of the Scala Collection EDSL.

5.5 Lifting collections

In this section, we first explain how for-comprehensions are desugared to calls to library functions, allowing an external library to give them a different meaning. We summarize afterwards needed information about the subset of the Scala collection EDSL that we reify. Then we present how we perform this reification. We finally present the reification of the running example (Fig. 3.1).

For-comprehensions As we have seen, an idiomatic encoding in Scala of queries on collections are for-comprehensions. Although Scala is an impure functional language and supports side-effectful for-comprehensions, only pure queries are supported in our framework, because this enables or simplifies many domain-specific analyses. Hence we will restrict our attention to pure queries.

The Scala compiler desugars for-comprehensions into calls to three collection methods, `map`, `flatMap` and `withFilter`, which we explain shortly; the query in Fig. 2.2 is desugared to the code in Fig. 5.1.

The compiler performs type inference and type checking on a for-comprehension only *after* desugaring it; this affords some freedom for the types of `map`, `flatMap` and `withFilter` methods.

The Scala Collection EDSL A Scala collection containing elements of type `T` implements the trait `Traversable[T]`. On an expression `coll` of type `Traversable[T]` one can invoke methods declared (in first approximation) as follows:

```
def map[U](f: T ⇒ U): Traversable[U]
def flatMap[U](f: T ⇒ Traversable[U]): Traversable[U]
def withFilter[U](p: T ⇒ Boolean): Traversable[T].
```

For a Scala collection `coll`, the expression `coll.map(f)` applies `f` to each element of `coll`, collects the results in a new collection and returns it; `coll.flatMap(f)` applies `f` to each element of `coll`, concatenates the results in a new collection and returns it; `coll.withFilter(p)` produces a collection containing the elements of `coll` which satisfy the predicate `p`.

However, Scala supports many different collection types, and this complicates the actual types of these methods. Each collection can further implement subtraits like `Seq[T] <: Traversable[T]` (for sequences), `Set[T] <: Traversable[T]` (for sets) and `Map[K, V] <: Traversable[(K, V)]` (for dictionaries); for each such trait, different implementations are provided.

One consequence of this syntactic desugaring is that a single for-comprehension can operate over different collection types. The type of the result depends essentially on the type of the root collection, that is books in the example above. The example above can hence be altered to produce a sequence rather than a set by simply converting the root collection to another type:

```
val recordsSeq = for {
   book ← books.toSeq
   if book.publisher == "Pearson Education"
   author ← book.authors
  } yield BookData(book.title, author.firstName + " " + author.lastName,
    book.authors.size - 1)
```

Precise static typing The Scala collections EDSL achieves precise static typing while avoiding code duplication [Odersky and Moors, 2009]. Precise static typing is necessary because the return type of a query operation can depend on subtle details of the base collection and query arguments. To return the most specific static type, the Scala collection DSL defines a type-level relation between the source collection type, the element type for the transformed collection, and the type for the resulting collection. The relation is encoded through the concept pattern [Oliveira et al., 2010], i.e., through a type-class-style trait `CanBuildFrom[From, Elem, To]`, and elements of the relation are expressed as implicit instances.

For example, a finite map can be treated as a set of pairs so that mapping a function from pairs to pairs over it produces another finite map. This behavior is encoded in an instance of type `CanBuildFrom[Map[K, V], (K1, V1), Map[K1, V1]`. The `Map[K, V]` is the type of the base collection, `(K1, V1)` is the return type of the function, and `Map[K1, V1]` is the return type of the map operation.

It is also possible to map some other function over the finite map, but the result will be a general collection instead of a finite map. This behavior is described by an instance of type `CanBuildFrom[Traversable[T], U, Traversable[U]`. Note that this instance is also applicable to finite maps, because `Map` is a subclass of `Traversable`. Together, these two instances describe how to compute the return type of mapping over a finite map.

Code reuse Even though these two use cases for mapping over a finite map have different return types, they are implemented as a single method that uses its implicit `CanBuildFrom` parameter to compute both the static type and the dynamic representation of its result. So the Scala Collections EDSL provides precise typing without code duplication. In our deep embedding, we want to preserve this property.

`CanBuildFrom` is used in the implementation of `map`, `flatMap` and `withFilter`. To further increase reuse, the implementations are provided in a helper trait `TraversableLike[T, Repr]`, with the following signatures:

```
def map[U](f: T ⇒ U)(implicit cbf: CanBuildFrom[Repr, U, That]): That
def flatMap[U](f: T ⇒ Traversable[U])
  (implicit cbf: CanBuildFrom[Repr, U, That]): That
def withFilter[U](p: T ⇒ Boolean): Repr.
```

The `Repr` type parameter represents the specific type of the receiver of the method call.

The lifting The basic idea is to use the enrich-my-library pattern to lift collection methods from `TraversableLike[T, Repr]` to `TraversableLikeOps[T, Repr]`:

```
implicit def expToTraversableLikeOps[T, Repr]
    (v: Exp[TraversableLike[T, Repr]]) =
  new TraversableLikeOps[T, Repr](v)
```

Subclasses of `TraversableLike[T, Repr]` also subclass both `Traversable[T]` and `Repr`; to take advantage of this subclass relation during interpretation and optimization, we need to restrict the

```
class TraversableLikeOps[T,
    Repr <: Traversable[T] with TraversableLike[T, Repr]](t: Exp[Repr]) {
  val t: Exp[Repr]

  def withFilter(f: Exp[T] ⇒ Exp[Boolean]): Exp[Repr] =
    Filter(this.t, FuncExp(f))

  def map[U, That <: TraversableLike[U, That]](f: Exp[T] ⇒ Exp[U])
        (implicit c: CanBuildFrom[Repr, U, That]): Exp[That] =
    Node(this.t, FuncExp(f))
  //...other methods...
}
//definitions of MapNode, Filter omitted.
```

Figure 5.2: Lifting `TraversableLike`

type of `expToTraversableLikeOps`, obtaining the following conversion:[5]

```
implicit def expToTraversableLikeOps[T,
    Repr <: Traversable[T] with TraversableLike[T, Repr]](v: Exp[Repr]) =
  new TraversableLikeOps[T, Repr](v)
```

The query operations are defined in class `TraversableLikeOps[T, Repr]`; a few examples are shown in Fig. 5.2.[6]

Note how the lifted query operations use `CanBuildFrom` to compute the same static return type as the corresponding non-lifted query operation would compute. This reuse of type-level machinery allows SQuOpt to provide the same interface as the Scala Collections EDSL.

Code reuse, revisited We already saw how we could reify `List[T].map` through a specific expression node, `ListMapNode`. However, this approach would require generating many variants for different collections with slightly different types; writing an optimizer able to handle all such variations would be unfeasible because of the amount of code duplication required. Instead, by reusing Scala type-level machinery, we obtain a reification which is statically typed and at the same time avoids code duplication in both our lifting and our optimizer, and in general in all possible consumers of our reification, making them feasible to write.

5.6 Interpretation

After optimization, SQuOpt needs to interpret the optimized expression trees to perform the query. Therefore, the trait `Exp[T]` declares a `def interpret(): T` method, and each expression node overrides it appropriately to implement a mostly standard typed, tagless [Carette et al., 2009], environment-based interpreter. The interpreter computes a value of type `T` from an expression tree of type `Exp[T]`. This design allows query writers to extend the interpreter to handle application-specific operations. In fact, the lifting generator described in Sec. 5.4 automatically generates appropriate definitions of `interpret` for the lifted operations.

[5] Due to type inference bugs, the actual implicit conversion needs to be slightly more complex, to mention `T` directly in the argument type. We reported the bug at https://issues.scala-lang.org/browse/SI-5298.

[6] Similar liftings are introduced for traits similar to `TraversableLike`, like `SeqLike`, `SetLike`, `MapLike`, and so on.

For example, the interpretation of string concatenation is simply string concatenation, as shown in the following fragment of the interpreter. Note that type-safety of the interpreter is checked automatically by the Scala compiler when it compiles the fragments.

```
case class StringConcat(str1: Exp[String], str2: Exp[String])
    extends Exp[String] {
  def interpret() = str1.interpret() + str2.interpret()
}
```

The subset of Scala we reify roughly corresponds to a typed lambda calculus with subtyping and type constructors. It does not include constructs for looping or recursion, so it should be strongly normalizing as long as application programmers do not add expression nodes with non-terminating interpretations. However, query writers can use the host language to create a reified expression of infinite size. This should not be an issue if SQUOPT is used as a drop-in replacement for the Scala Collection EDSL.

During optimization, nodes of the expression tree might get duplicated, and the interpreter could, in principle, observe this sharing and treat the expression tree as a DAG, to avoid recomputing results. Currently, we do not exploit this, unlike during compilation.

5.7 Optimization

Our optimizer is structured as a pipeline of different transformations on a single intermediate representation, constituted by our expression trees. Each phase of the pipeline, and the pipeline as a whole, produce a new expression having the same type as the original one. Most of our transformations express simple rewrite rules with or without side conditions, which are applied on expression trees from the bottom up and are implemented using Scala's support for pattern matching [Emir et al., 2007b].

Some optimizations, like filter hoisting (which we applied manually to produce the code in Fig. 2.3), are essentially domain-specific and can improve complexity of a query. To enable such optimizations to trigger, however, one needs often to perform inlining-like transformations and to simplify the result. Inlining-related transformation can for instance produce code like `(x, y)._1`, which we simplify to `x`, reducing abstraction overhead but also (more importantly) making syntactically clear that the result does not depend on `y`, hence might be computed before `y` is even bound. This simplification extends to user-defined product types; with definitions in Fig. 2.1 code like `BookData(book.title, ... ).title` is simplified to `book.title`.

We have implemented thus optimizations of three classes:

- general-purpose simplifications, like inlining, compile-time beta-reduction, constant folding and reassociation on primitive types, and other simplifications[7];
- domain-specific simplifications, whose main goal is still to enable more important optimizations;
- domain-specific optimizations which can change the complexity class of a query, such as filter hoisting, hash-join transformation or indexing.

Among domain-specific simplifications, we implement a few described in the context of the monoid comprehension calculus [Grust and Scholl, 1996b; Grust, 1999], such as query unnesting and fusion of bulk operators. Query unnesting allows to *unnest* a for comprehension nested inside

[7] Beta-reduction and simplification are run in a fixpoint loop [Peyton Jones and Marlow, 2002]. Termination is guaranteed because our language does not admit general recursion.

another, and produce a single for-comprehension. Furthermore, we can fuse different collection operations together: collection operators like `map`, `flatMap` and `withFilter` can be expressed as folds producing new collections which can be combined. Scala for-comprehension are however more general than monoid comprehensions[8], hence to ensure safety of some optimizations we need some additional side conditions[9].

Manipulating functions To be able to inspect a HOAS function body `funBody: Exp[S] ⇒ Exp[T]`, like `str ⇒ str + "!"`, we convert it to *first-order* abstract syntax (FOAS), that is to an expression tree of type `Exp[T]`. To this end, we introduce a representation of variables and a generator of fresh ones; since variable names are auto-generated, they are internally represented simply as integers instead of strings for efficiency.

To convert `funBody` from HOAS to FOAS we apply it to a fresh variable `v` of type `TypedVar[S]`, obtaining a first-order representation of the function body, having type `Exp[T]`, and containing occurrences of `v`.

This transformation is hidden into the constructor `Fun`, which converts `Exp[S] ⇒ Exp[T]`, a representation of an expression with one free variable, to `Exp[S ⇒ T]`, a representation of a function.

```
case class App[S, T](f: Exp[S ⇒ T], t: Exp[S]) extends Exp[T]
def Fun[-S, +T](f: Exp[S] ⇒ Exp[T]): Exp[S ⇒ T] = {
  val v = Fun.gensym[S]()
  FOASFun(funBody(v), v)
}
case class FOASFun[S, T](val foasBody: Exp[T], v: TypedVar[S])
  extends Exp[S ⇒ T]
implicit def app[S, T](f: Exp[S ⇒ T]): Exp[S] ⇒ Exp[T] =
  arg ⇒ App(f, arg)
```

Conversely, function applications are represented using the constructor `App`; an implicit conversion allows `App` to be inserted implicitly. Whenever `f` can be applied to `arg` and `f` is an expression tree, the compiler will convert `f(arg)` to `app(f)(arg)`, that is `App(f, arg)`.

In our example, `Fun(str ⇒ str + "!")` produces

```
FOASFun(StringConcat(TypedVar[String](1), Const("!")),
  TypedVar[String](1))
```

Since we auto-generate variable names, it is easiest to implement represent variable occurrences using the Barendregt convention, where bound variables must always be globally unique; we must be careful to perform renaming after beta-reduction to restore this invariant [Pierce, 2002, Ch. 6].

We can now easily implement substitution and beta-reduction and through that, as shown before, enable other optimizations to trigger more easily and speedup queries.

[8] For instance, a for-comprehension producing a list cannot iterate over a set.

[9] For instance, consider a for-comprehension producing a set and nested inside another producing a list. This comprehension does not correspond to a valid monoid comprehension (see previous footnote), and query unnesting does not apply here: if we unnested the inner comprehension into the outer one, we would not perform duplicate elimination on the inner comprehension, affecting the overall result.

Chapter 6

Evaluating SQuOpt

The key goals of SQuOpt are to reconcile *modularity* and *efficiency*. To evaluate this claim, we perform a rigorous performance evaluation of queries with and without SQuOpt. We also analyze modularization potential of these queries and evaluate how modularization affects performance (with and without SQuOpt).

We show that modularization introduces a significant slowdown. The overhead of using SQuOpt is usually moderate, and optimizations can compensate this overhead, remove the modularization slowdown and improve performance of some queries by orders of magnitude, especially when indexes are used.

6.1 Study Setup

Throughout this chapter, we have already shown several compact queries for which our optimizations increase performance significantly compared to a naive execution. Since some optimizations change the complexity class of the query (e.g. by using an index), so the speedups grow with the size of the data. However, to get a more realistic evaluation of SQuOpt, we decided to perform an experiment with existing real-world queries.

As we are interested in both performance and modularization, we have a specification and three different implementations of each query that we need to compare:

(0) **Query specification:** We selected a set of existing real-world queries specified and implemented independently from our work and prior to it. We used only the specification of these queries.

(1) **Modularized Scala implementation:** We reimplemented each query as an expression on Scala collections — our baseline implementation. For modularity, we separated reusable domain abstractions into subqueries. We confirmed the abstractions with a domain expert and will later illustrate them to emphasize their general nature.

(2) **Hand-optimized Scala implementation:** Next, we asked a domain expert to performed manual optimizations on the modularized queries. The expert should perform optimizations, such as inlining and filter hoisting, where he could find performance improvements.

(3) **SQuOpt implementation:** Finally, we rewrote the modularized Scala queries from (1) as SQuOpt queries. The rewrites are of purely syntactic nature to use our library (as described in Sec. 3.1) and preserve the modularity of the queries.

Since SQuOpt supports executing queries with and without optimizations and indexes, we measured actually three different execution modes of the SQuOpt implementation:

(3^-) **SQuOpt without optimizer:** First, we execute the SQuOpt queries without performing optimization first, which should show the SQuOpt overhead compared to the modular Scala implementation (1). However, common-subexpression elimination is still used here, since it is part of the compilation pipeline. This is appropriate to counter the effects of excessive inlining due to using a deep embedding, as explained in Sec. 4.1.

(3^o) **SQuOpt with optimizer:** Next, we execute SQuOpt queries after optimization.

(3^x) **SQuOpt with optimizer and indexes:** Finally, we execute the queries after providing a set of indexes that the optimizer can consider.

In all cases, we measure query execution time for the generated code, excluding compilation: we consider this appropriate because the results of compilations are cached aggressively and can be reused when the underlying data is changed, potentially even across executions (even though this is not yet implemented), as the data is not part of the compiled code.

We use additional indexes in (3^x), but not in the hand-optimized Scala implementation (2). We argue that indexes are less likely to be applied manually, because index application is a crosscutting concern and makes the whole query implementation more complicated and less abstract. Still, we offer measurement (3^o) to compare the speedup without additional indexes.

This gives us a total of five settings to measure and compare (1, 2, 3^-, 3^o, and 3^x). Between them, we want to observe the following interesting performance ratios (speedups or slowdowns, computed through the indicated divisions):

(M) Modularization overhead (the relative performance difference between the modularized and the hand-optimized Scala implementation: $1/2$).

(S) SQuOpt overhead (the overhead of executing unoptimized SQuOpt queries: $1/3^-$; smaller is better).

(H) Hand-optimization challenge (the performance overhead of our optimizer against hand-optimizations of a domain expert: $2/3^o$; bigger is better). This overhead is partly due to the SQuOpt overhead (S) and partly to optimizations which have not been automated or have not been effective enough. This comparison excludes the effects of indexing, since this is an optimization we did not perform by hand; we also report **(H')** = $2/3^x$, which includes indexing.

(O) Optimization potential (the speedup by optimizing modularized queries: $1/3^o$; bigger is better).

(X) Index influence (the speedup gained by using indexes: $3^o/3^x$) (bigger is better).

(T) Total optimization potential with indexes ($1/3^x$; bigger is better), which is equal to $(O) \times (X)$.

In Fig. 6.1, we provide an overview of the setup. We made our raw data available and our results reproducible [Vitek and Kalibera, 2011].[1]

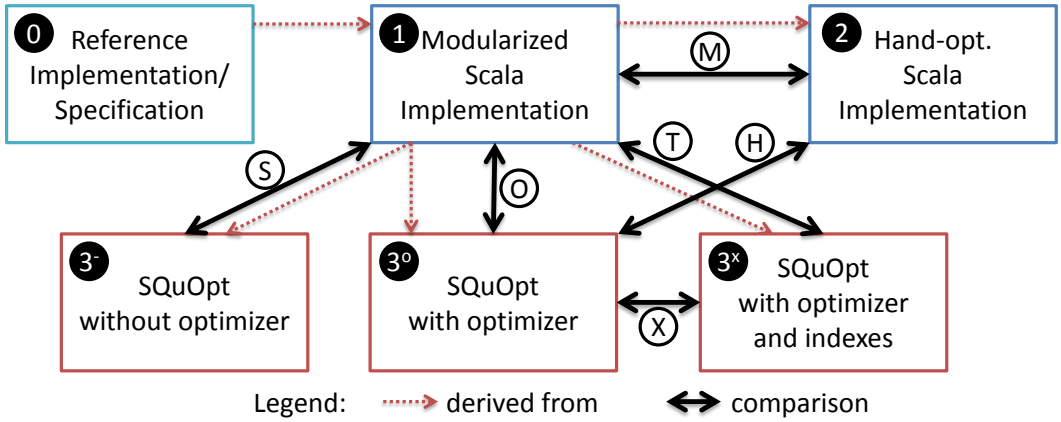


Figure 6.1: Measurement Setup: Overview

6.2 Experimental Units

As experimental units, we sampled a set of queries on code structures from FindBugs 2.0 [Hovemeyer and Pugh, 2004]. FindBugs is a popular bug-finding tool for Java Bytecode available as open source. To detect instances of bug patterns, it queries a structural in-memory representation of a code base (extracted from bytecode). Concretely, a single loop traverses each class and invokes all visitors (implemented as listeners) on each element of the class. Many visitors, in turn, perform activities concerning multiple bug detectors which are fused together. An extreme example is that, in FindBugs, query 4 is defined in class `DumbMethods` together with other 41 bug detectors for distinct types of bugs. Typically a bug detector is furthermore scattered across the different methods of the visitor, which handle different elements of the class. We believe this architecture has been chosen to achieve good performance; however, we do not consider such manual fusion of distinct bug detectors together as modular. We selected queries from FindBugs because they represent typical non-trivial queries on in-memory collections and because we believe our framework allows expressing them more modularly.

We sampled queries in two batches. First, we manually selected 8 queries (from approx. 400 queries in FindBugs), chosen mainly to evaluate the potential speedups of indexing (queries that primarily looked for declarations of classes, methods, or fields with specific properties, queries that inspect the type hierarchy, and queries that required analyzing methods implementation). Subsequently, we *randomly* selected a batch of 11 additional queries. The batch excluded queries that rely on control-/dataflow analyses (i.e., analyzing the effect of bytecode instructions on the stack), due to limitations of the bytecode tookit we use. In total, we have 19 queries as listed in Table 6.2 (the randomly selected queries are marked with the superscript R).

We implemented each query three times (see implementations (1)–(3) in Sec. 6.1) following the specifications given in the FindBugs documentation (0). Instead of using a hierarchy of visitors as the original implementations of the queries in FindBugs, we wrote the queries as for-comprehensions in Scala on an in-memory representation created by the Scala toolkit BAT.[2] BAT in particular provides comprehensive support for writing queries against Java bytecode in an idiomatic way. We exemplify an analysis in Fig. 6.5: It detects all co-variant `equals` methods in a project by iterating over all class files (line 2) and all methods, searching for methods named "`equals`" that return a boolean value

[1]Data available at: http://www.informatik.uni-marburg.de/~pgiarrusso/SQuOpt
[2]http://github.com/Delors/BAT

		Performance (ms)					Performance ratios		
Id	Description	1	2	3^{-}	3^{o}	3^{x}	M (1/2)	H ($2/3^{o}$)	T ($1/3^{x}$)
1	Covariant compareTo() defined	1.1	1.3	0.85	0.26	0.26	0.9	5.0	4.4
2	Explicit garbage collection call	496	258	1176	1150	52	1.9	0.2	9.5
3	Protected field in final class	11	1.1	11	1.2	1.2	10.0	1.0	9.8
4	Explicit runFinalizersOnExit() call	509	262	1150	1123	10.0	1.9	0.2	51
5	clone() defined in non-Cloneable class	29	14	55	46	0.47	2.1	0.3	61
6	Covariant equals() defined	29	15	23	9.7	0.20	1.9	1.6	147
7	Public finalizer defined	29	12	28	8.0	0.03	2.3	1.5	1070
8	Dubious catching of IllegalMonitorStateException	82	72	110	28	0.01	1.1	2.6	12800
9^{R}	Uninit. field read during construction of super	896	367	3017	960	960	2.4	0.4	0.9
10^{R}	Mutable static field declared public	9527	9511	9115	9350	9350	1.0	1.0	1.0
11^{R}	Refactor anon. inner class to static	8804	8767	8718	8700	8700	1.0	1.0	1.0
12^{R}	Inefficient use of toArray(Object[])	3714	1905	4046	3414	3414	2.0	0.6	1.1
13^{R}	Primitive boxed and unboxed for coercion	3905	1672	5044	3224	3224	2.3	0.5	1.2
14^{R}	Double precision conversion from 32 bit	3887	1796	5289	3010	3010	2.2	0.6	1.3
15^{R}	Privileged method used outside doPrivileged	505	302	1319	337	337	1.7	0.9	1.5
16^{R}	Mutable public static field should be final	13	6.2	12	7.0	7.0	2.0	0.9	1.8
17^{R}	Serializable class is member of non-ser. class	12	0.77	0.94	1.8	1.8	16	0.4	6.9
18^{R}	Swing methods used outside Swing thread	577	53	1163	45	45	11	1.2	13
19^{R}	Finalizer only calls super class finalize	55	13	73	11	0.10	4.4	1.1	541

Table 6.2: Performance results. As in in Sec. 6.1, (1) denotes the modular Scala implementation, (2) the hand-optimized Scala one, and (3^{-}), (3^{o}), (3^{x}) refer to the SQuOpt implementation when run, respectively, without optimizations, with optimizations, with optimizations and indexing. Queries marked with the R superscript were selected by random sampling.

	M (1/2)	S ($1/3^-$)	H ($2/3^o$)	H' ($2/3^x$)	O ($1/3^o$)	X ($3^o/3^x$)	T ($1/3^x$)
Geometric means of performance ratios	2.4x	1.2x	0.8x	5.1x	1.9x	6.3x	12x

Table 6.3: Average performance ratios. This table summarizes all interesting performance ratios across all queries, using the geometric mean [Fleming and Wallace, 1986]. The meaning of speedups is discussed in Sec. 6.1.

Abstraction	Used
All fields in all class files	4
All methods in all class files	3
All method bodies in all class files	3
All instructions in all method bodies and their bytecode index	5
Sliding window (size n) over all instructions (and their index)	3

Table 6.4: Description of abstractions removed during hand-optimization and number of queries where the abstraction is used (and optimized away).

and define a single parameter of the type of the current class.

Abstractions In the reference implementations (1), we identified several reusable abstractions as shown in Table 6.4. The reference implementations of all queries except 17^R use exactly one of these abstractions, which encapsulate the main loops of the queries.

Indexes For executing (3^x) (SQuOpt with indexes), we have constructed three indexes to speed up navigation over the queried data of queries 1–8: Indexes for method name, exception handlers, and instruction types. We illustrate the implementation of the method-name index in Fig. 6.6: it produces a collection of all methods and then indexes them using `indexBy`; its argument extracts from an entry the key, that is the method name. We selected which indexes to implement using guidance from SQuOpt itself; during optimizations, SQuOpt reports which indexes it could have applied to the given query. Among those, we tried to select indexes giving a reasonable compromise between construction cost and optimization speedup. We first measured the construction cost of these indexes:

Index	Elapsed time (ms)
Method name	97.99±2.94
Exception handlers	179.29±3.21
Instruction type	4166.49±202.85

For our test data, index construction takes less than 200 ms for the first two indexes, which is moderate compared to the time for loading the bytecode in the BAT representation (4755.32 ± 141.66). Building the instruction index took around 4 seconds, which we consider acceptable since this index maps each type of instruction (e.g. `INSTANCEOF`) to a collection of all bytecode instructions of that type.

6.3 Measurement Setup

To measure performance, we executed the queries on the preinstalled JDK class library (`rt.jar`), containing 58M of uncompressed Java bytecode. We also performed a preliminary evaluation by

```
for {
  classFile ← classFiles.asSquopt
  method ← classFile.methods
  if method.isAbstract && method.name ==# "equals" &&
     method.descriptor.returnType ==# BooleanType
  parameterTypes ← Let(method.descriptor.parameterTypes)
  if parameterTypes.length ==# 1 &&
     parameterTypes(0) ==# classFile.thisClass
} yield (classFile, method)
```

Figure 6.5: Find covariant equals methods.

```
val methodNameIdx: Exp[Map[String, Seq[(ClassFile, Method)]]] = (for {
  classFile ← classFiles.asSquopt
  method ← classFile.methods
} yield (classFile, method)).indexBy(entry ⇒ entry._2.name)
```

Figure 6.6: A simple index definition

running queries on the much smaller ScalaTest library, getting comparable results that we hence do not discuss. Experiments were run on a 8-core Intel Core i7-2600, 3.40 GHz, with 8 GB of RAM, running Scientific Linux release 6.2. The benchmark code itself is single-threaded, so it uses only one core; however the JVM used also other cores to offload garbage collection. We used the preinstalled OpenJDK Java version 1.7.0_05-icedtea and Scala 2.10.0-M7.

We measure steady-state performance as recommended by Georges et al. [2007]. We invoke the JVM $p = 15$ times; at the beginning of each JVM invocation, all the bytecode to analyze is loaded in memory and converted into BAT's representation. In each JVM invocation, we iterate each benchmark until the variations of results becomes low enough. We measure the variations of results through the coefficient of variation (CoV; standard deviation divided by the mean). Thus, we iterate each benchmark until the CoV in the last $k = 10$ iterations drops under the threshold $\theta = 0.1$, or until we complete $q = 50$ iterations. We report the arithmetic mean of these measurements (and also report the usually low standard deviation on our web page).

6.4 Results

Correctness We machine-checked that for each query, all variants in Table 6.2 agree.

Modularization Overhead We first observe that performance suffers significantly when using the abstractions we described in Table 6.4. These abstractions, while natural in the domain and in the setting of a declarative language, are not idiomatic in Java or Scala because, without optimization, they will obviously lead to bad performance. They are still useful abstractions from the point of view of modularity, though — as indicated by Table 6.4 — and as such it would be desirable if one could use them without paying the performance penalty.

Scala Implementations vs. FindBugs Before actually comparing between the different Scala and SQuOpt implementations, we first ensured that the implementations are comparable to the original FindBugs implementation. A direct comparison between the FindBugs reference implementation and any of our implementations is not possible in a rigorous and fair manner. FindBugs bug detectors are not fully modularized, therefore we cannot reasonably isolate the implementation

of the selected queries from support code. Furthermore, the architecture of the implementation has many differences that affect performance: among others, FindBugs also uses multithreading. Moreover, while in our case each query loops over all classes, in FindBugs, as discussed above, a single loop considers each class and invokes all visitors (implemented as listeners) on it.

We measured *startup performance* [Georges et al., 2007], that is the performance of running the queries only once, to minimize the effect of compiler optimizations. We setup our SQuOpt-based analyses to only perform optimization and run the optimized query. To setup FindBugs, we manually disabled all unrelated bug detectors; we also made the modified FindBugs source code available. The result is that the performance of the Scala implementations of the queries (3^-) has performance of the same order of magnitude as the original FindBugs queries – in our tests, the SQuOpt implementation was about twice as fast. However, since the comparison cannot be made fair, we refrained from a more detailed investigation.

SQuOpt Overhead and Optimization Potential We present the results of our benchmarks in Table 6.2. Column names refer to a few of the definitions described above; for readability, we do not present all the ratios previously introduced for each query, but report the raw data. In Table 6.3, we report the geometric mean Fleming and Wallace [1986] of each ratio, computed with the same weight for each query.

We see that, in its current implementation, SQuOpt can cause a overhead S ($1/3^-$) up to 3.4x. On average SQuOpt queries are 1.2x faster. These differences are due to minor implementation details of certain collection operators. For query 18^R, instead, we have that the the basic SQuOpt implementation is 12.9x faster and are investigating the reason; we suspect this might be related to the use of pattern matching in the original query.

As expected, not all queries benefit from optimizations; out of 19 queries, optimization affords for 15 of them significant speedups ranging from a 1.2x factor to a 12800x factor; 10 queries are faster by a factor of at least 5. Only queries 10^R, 11^R and 12^R fail to recover any modularization overhead.

We have analyzed the behavior of a few queries after optimization, to understand why their performance has (or has not) improved.

Optimization makes query 17^R slower; we believe this is because optimization replaces filtering by lazy filtering, which is usually faster, but not here. Among queries where indexing succeeds, query 2 has the least speedup. After optimization, this query uses the instruction-type index to find all occurrences of invocation opcodes (`INVOKESTATIC` and `INVOKEVIRTUAL`); after this step the query looks, among those invocations, for ones targeting `runFinalizersOnExit`. Since invocation opcodes are quite frequent, the used index is not very specific, hence it allows for little speedup (9.5x). However no other index applies to this query; moreover, our framework does not maintain any selectivity statistics on indexes to predict these effects. Query 19^R benefits from indexing without any specific tuning on our part, because it looks for implementations of `finalize` with some characteristic, hence the highly selective method-name index applies. After optimization, query 8 becomes simply an index lookup on the index for exception handlers, looking for handlers of `IllegalMonitorStateException`; it is thus not surprising that its speedup is thus extremely high (12800x). This speedup relies on an index which is specific for this kind of query, and building this index is slower than executing the unoptimized query. On the other hand, building this index is entirely appropriate in a situation where similar queries are common enough. Similar considerations apply to usage of indexing in general, similarly to what happens in databases.

Optimization Overhead The current implementation of the optimizer is not yet optimized for speed (of the optimization algorithm). For instance, expression trees are traversed and rebuilt completely once for each transformation. However, the optimization overhead is usually not excessive and is 54.8 ± 85.5 ms, varying between 3.5 ms and 381.7 ms (mostly depending on the query size).

Limitations Although many speedups are encouraging, our optimizer is currently a proof-of-concept and we experienced some limitations:

- In a few cases hand-optimized queries are still faster than what the optimizer can produce. We believe these problems could be addressed by adding further optimizations.
- Our implementation of indexing is currently limited to immutable collections. For mutable collections, indexes must be maintained incrementally. Since indexes are defined as special queries in SQuOpt, incremental index maintenance becomes an instance of incremental maintenance of query results, that is, of incremental view maintenance. We plan to support incremental view maintenance as part of future work; however, indexing in the current form is already useful, as illustrated by our experimental results.

Threats to Validity With rigorous performance measurements and the chosen setup, our study was setup to maximize internal and construct validity. Although we did not involve an external domain expert and we did not compare the results of our queries with the ones from FindBugs (except while developing the queries), we believe that the queries adequately represent the modularity and performance characteristics of FindBugs and SQuOpt. However, since we selected only queries from a single project, external validity is limited. While we cannot generalize our results beyond FindBugs yet, we believe that the FindBugs queries are representative for complex in-memory queries performed by applications.

Summary We demonstrated on our real-world queries that relying on declarative abstractions in collection queries often causes a significant slowdown. As we have seen, using SQuOpt without optimization, or when no optimizations are possible, usually provides performance comparable to using standard Scala; however, SQuOpt optimizations can in most cases remove the slowdown due to declarative abstractions. Furthermore, relying on indexing allows to achieve even greater speedups while still using a declarative programming style. Some implementation limitations restrict the effectiveness of our optimizer, but since this is a preliminary implementation, we believe our evaluation shows the great potential of optimizing queries to in-memory collections.

Chapter 7

Discussion

In this chapter we discuss the degree to which SQuOpt fulfilled our original design goals, and the conclusions for host and domain-specific language design.

7.1 Optimization limitations

In our experiments indexing achieved significant speedups, but when indexing does not apply speedups are more limited; in comparison, later projects working on collection query optimization, such as OptiQL [Rompf et al., 2013; Sujeeth et al., 2013b], gave better speedups, as also discussed in Sec. 8.3.1.

A design goal of this project was to incrementalize optimized queries, and while it is easy to incrementalize collection operators such as `map`, `flatMap` or `filter`, it was much less clear to us how to optimize the result of inlining. We considered using shortcut fusion, but we did not know a good way to incrementalize the resulting programs; later work, as described in Part II, clarified what is possible and what not.

Another problem is that most fusion techniques are designed for sequential processing, hence conflict with incrementalization. Most fusion techniques generally assume bulk operations scan collections in linear order. For instance, shortcut fusion rewrites operators in terms of `foldr`. During parallel and/or incremental computation, instead, it is better to use *tree-shaped folds*: that is, to split the task in a divide-and-conquer fashion, so that the various subproblems form a balanced tree. This division minimizes the height of the tree, hence the number of steps needed to combine results from subproblems into the final result, as also discussed by Steele [2009]. It is not so clear how to apply shortcut fusion to parallel and incremental programs. Maier and Odersky [2013] describe an incremental tree-shaped fold, where each subproblem that is small enough is solved by scanning its input in linear order, but does not perform code transformation and does not study how to perform fusion.

7.2 Deep embedding

7.2.1 What worked well

Several features of Scala contributed greatly to the success we achieved. With implicit conversions, the lifting can be made mostly transparent. The advanced type system features were quite helpful to make the expression tree representation typed. The fact that for-comprehensions are desugared

before type inference and type checking was also a prerequisite for automatic lifting. The syntactic expressiveness and uniformity of Scala, in particular the fact that custom types can have the same look-and-feel as primitive types, were also vital to lift expressions on primitive types.

7.2.2 Limitations

Despite these positive experiences and our experimental success, our embedding has a few significant limitations.

The first limitation is that we only lift a subset of Scala, and some interesting features are missing. We do not support *statements* in nested blocks in our queries, but this could be implemented reusing techniques from Delite [Rompf et al., 2011]. More importantly for queries, *pattern matching* cannot be supported by deep embedding similar to ours. In contrast to for-comprehension syntax, pattern matching is desugared only *after* type checking Emir et al. [2007b], which prevents us from lifting pattern matching notation. More specifically, because an extractor Emir et al. [2007b] cannot return the representation of a result value (say `Exp[Boolean]`) to later evaluate; it must produce its final result at pattern matching time. There is initial work on "virtualized pattern matching"[1], and we hope to use this feature in future work.

We also experienced problems with operators that cannot be overloaded, such as == or **`if-else`** and with lifting methods in `scala.Any`, which forced us to provide alternative syntax for these features in queries. The Scala-virtualized project [Moors et al., 2012] aims to address these limitations; unfortunately, it advertises no change on the other problems we found, which we subsequently detail.

It would also be desirable if we could enforce the absence of side effects in queries, but since Scala, like most practical programming languages except Haskell, does not track side effects in the type system this does not seem to be possible.

Finally, compared to *lightweight modular staging* [Rompf and Odersky, 2010] (the foundation of Delite) and to polymorphic embedding [Hofer et al., 2008], we have less static checking for some programming errors when writing queries; the recommended way to use Delite is to write a EDSL program in one trait, in terms of the EDSL interface only, and combine it with the implementation in another trait. In polymorphic embedding, the EDSL program is a function of the specific implementation (in this case, semantics). Either approach ensures that the DSL program is *parametric* in the implementation, and hence cannot refer to details of a specific implementation. However, we judged the syntactic overhead for the programmer to be too high to use those techniques – in our implementation we rely on encapsulation and on dynamic checks at query construction time to achieve similar guarantees.

The choice of interpreting expressions turned out to be a significant performance limitation. It could likely be addressed by using Delite and lightweight modular staging instead, but we wanted to experiment with how far we can get *within* the language in a well-typed setting.

7.2.3 What did not work so well: Scala type inference

When implementing our library, we often struggled against limitations and bugs in the Scala compiler, especially regarding type inference and its interaction with implicit resolution, and we were often constrained by its limitations. Not only Scala's type inference is not complete, but we learned that its behavior is only specified by the behavior of the current implementation: in many cases where there is a clear desirable solution, type inference fails or finds an incorrect substitution which leads to a type error. Hence we cannot distinguish, in the discussion, the Scala language from its implementation. We regard many of Scala's type inference problems as bugs, and reported them

[1] http://stackoverflow.com/questions/8533826/what-is-scalas-experimental-virtual-pattern-matcher

as such when no previous bug report existed, as noted in the rest of this section. Some of them are long-standing issues, others of them were accepted, for other ones we received no feedback yet at the time of this writing, and another one was already closed as WONTFIX, indicating that a fix would be possible but have excessive complexity for the time being.[2]

Overloading The code in Fig. 3.1 uses the lifted `BookData` constructor. Two definitions of `BookData` are available, with signatures `BookData(String, String, Int)` and `BookData(Exp[String], Exp[String], Exp[Int])`, and it seems like the Scala compiler should be able to choose which one to apply using overload resolution. This however is not supported simply because the two functions are defined in different scopes,[3] hence importing `BookData(Exp[String], Exp[String], Exp[Int])` shadows locally the original definition.

Type inference vs implicit resolution The interaction between type inference and implicit resolution is a hard problem, and Scalac has also many bugs, but the current situation requires further research; for instance, there is not even a specification for the behavior of type inference[4].

As a consequence, to the best of our knowledge some properties of type inference have not been formally established. For instance, a reasonable user expectation is that removing a call to an implicit conversion does not alter the program, if it is the only implicit conversion with the correct type in scope, or if it is more specific than the others [Odersky et al., 2011, Ch. 21]. This is not always correct, because removing the implicit conversion reduces the information available for the type inference algorithm; we observed multiple cases[5] where type inference becomes unable to figure out enough information about the type to trigger implicit conversion.

We also consider significant that Scala 2.8 required making both type inference and implicit resolution more powerful, specifically in order to support the collection library [Moors, 2010; Odersky et al., 2011, Sec. 21.7]; further extensions would be possible and desirable. For instance, if type inference were extended with higher-order unification[6] [Pfenning, 1988], it would better support a part of our DSL interface (not discussed in this chapter) by removing the need for type annotations.

Nested pattern matches for GADTs in Scala Writing a typed decomposition for `Exp` requires pattern-matching support for generalized algebraic datatypes (GADTs). We found that support for GADTs in Scala is currently insufficient. Emir et al. [2007b] define the concept of *typecasing*, essentially a form of pattern-matching limited to non-nested patterns, and demonstrate that Scala supports typecasing on GADTs in Scala by demonstrating a typed evaluator; however, typecasing is rather verbose for deep patterns, since one has to nest multiple pattern-matching expressions. When using normal pattern matches, instead, the support for GADT seems much weaker.[7] Hence one has to choose between support for GADT and the convenience of nested pattern matching. A third alternative is to ignore or disable compiler warnings, but we did not consider this option.

Implicit conversions do not chain While implicit conversions by default do not chain, it is sometimes convenient to allow chaining selectively. For instance, let us assume a context such that `a: Exp[A]`, `b: Exp[B]` and `c: Exp[C]`. In this context, let us consider again how we lift tuples. We have seen that the expression `(a, b)` has type `(Exp[A], Exp[B])` but can be converted to `Exp[(A, B)]` through an implicit conversion. Let us now consider *nested* tuples, like `((a, b), c)`: it has type `((Exp[A], Exp[B]), Exp[C])`, hence the previous conversion cannot be applied to this expression.

Odersky et al. [2011, Ch. 21] describe a pattern which can address this goal. Using this pattern, to lift pairs, we must write an implicit conversion from pairs of elements which can be *implicitly converted* to expressions. Instead of requiring `(Exp[A], Exp[B])`, the implicit conversion should

[2] https://issues.scala-lang.org/browse/SI-2551
[3] https://issues.scala-lang.org/browse/SI-2551
[4] https://issues.scala-lang.org/browse/SI-5298?focusedCommentId=55971#comment-55971, reported by us.
[5] https://issues.scala-lang.org/browse/SI-5592, reported by us.
[6] https://issues.scala-lang.org/browse/SI-2712
[7] Due to bug https://issues.scala-lang.org/browse/SI-5298?focusedCommentId=56840#comment-56840, reported by us.

require `(A, B)` with the condition that `A` can be converted to `Exp[A']` and `B` to `Exp[B']`. This conversion solves the problem if applied explicitly, but the compiler refuses to apply it implicitly, again because of type inference issues[8].

Because of these type inference limitations, we failed to provide support for reifying code like `((a, b), c)`[9].

Error messages for implicit conversions The enrich-my-library pattern has the declared goal to allow to extend existing libraries *transparently*. However, implementation details shine however through when a user program using this feature contains a type error. When invoking a method would require an implicit conversion which is not applicable, the compiler often just reports that the method is not available. The recommended approach to debugging such errors is to manually add the missing implicit conversion and investigating the type error [Odersky et al., 2011, Ch. 21.8], but this destroys the transparency of the approach when creating or modifying code. We believe this could be solved in principle by research on error reporting: the compiler could automatically insert all implicit conversions enabling the method calls and report corresponding errors, even if at some performance cost.

7.2.4 Lessons for language embedders

Various domains, such as the one considered in our case study, allow powerful domain-specific optimizations. Such optimizations often are hard to express in a compositional way, hence they cannot be performed while building the query but must be expressed as global optimizations passes. For those domains, deep embedding is key to allow significant optimizations. On the other hand, deep embedding requires to implement an interpreter or a compiler.

On the one hand, interpretation overhead is significant in Scala, even when using HOAS to take advantage of the metalevel implementation of argument access.

Instead of interpreting a program, one can compile a EDSL program to Scala and load it, as done by Rompf et al. [2011]; while we are using this approach, the disadvantage is the compilation delay, especially for Scala whose compilation process is complex and time-consuming. Possible alternatives include generating bytecode directly or combining interpretation and compilations similarly to tiered JIT compilers, where only code which is executed often is compiled. We plan to investigate such alternatives in future work.

[8] https://issues.scala-lang.org/browse/SI-5651, reported by us.

[9] One could of course write a specific implicit conversions for *this* case; however, `(a, (b, c))` requires already a different implicit conversion, and there are infinite ways to nest pairs, let alone tuples of bounded arity.

Chapter 8

Related work

This chapter builds on prior work on language-integrated queries, query optimization, techniques for DSL embedding, and other works on code querying.

8.1 Language-Integrated Queries

Microsoft's Language-Integrated Query technology (LINQ) [Meijer et al., 2006; Bierman et al., 2007] is similar to our work in that it also reifies queries on collections to enable analysis and optimization. Such queries can be executed against a variety of backends (such as SQL databases or in-memory objects), and adding new back-ends is supported. Its implementation uses *expression trees*, a compiler-supported implicit conversion between expressions and their reification as a syntax tree. There are various major differences, though. First, the support for expression trees is hard-coded into the compiler. This means that the techniques are not applicable in languages that do not explicitly support expression trees. More importantly, the way expression trees are created in LINQ is generic and fixed. For instance, it is not possible to create different tree nodes for method calls that are relevant to an analysis (such as the `map` method) than for method calls that are irrelevant for the analysis (such as the `toString` method). For this reason, expression trees in LINQ cannot be customized to the task at hand and contain too much low-level information. It is well-known that this makes it quite hard to implement programs operating on expression trees [Eini, 2011].

LINQ queries can also not easily be decomposed and modularized. For instance, consider the task of refactoring the filter in the query `from x in y where x.z == 1 select x` into a function. Defining this function as `bool comp(int v) { return v == 1; }` would destroy the possibility of analyzing the filter for optimization, since the resulting expression tree would only contain a reference to an opaque function. The function could be declared as returning an expression tree instead, but then this function could not be used in the original query anymore, since the compiler expects an expression of type `bool` and not an expression tree of type `bool`. It could only be integrated if the expression tree of the original query is created by hand, without using the built-in support for expression trees.

Although queries against in-memory collections could theoretically also be optimized in LINQ, the standard implementation, LINQ2Objects, performs no optimizations.

A few optimized embedded DSLs allow executing queries or computations on distributed clusters. DryadLINQ [Yu et al., 2008], based on LINQ, optimizes queries for distributed execution. It inherits LINQ's limitations and thus does not support decomposing queries in different modules. Modularizing queries is supported instead by FlumeJava [Chambers et al., 2010], another library (in Java) for distributed query execution. However, FlumeJava cannot express many optimizations because its

representation of expressions is more limited; also, its query language is more cumbersome. Both problems are rooted in Java's limited support for embedded DSLs. Other embedded DSLs support parallel platforms such as GPUs or many-core CPUs, such as Delite [Rompf et al., 2013].

Willis et al. [2006, 2008] add first-class queries to Java through a source-to-source translator and implement a few selected optimizations, including join order optimization and incremental maintenance of query results. They investigate how well their techniques apply to Java programs, and they suggest that programmers use manual optimizations to avoid expensive constructs like nested loops. While the goal of these works is similar to ours, their implementation as an external source-to-source-translator makes the adoption, extensibility, and composability of their technique difficult.

There have been many approaches for a closer integration of SQL queries into programs, such as HaskellDB [Leijen and Meijer, 1999] (which also inspired LINQ), or Ferry [Grust et al., 2009] (which moves part of a program execution to a database). In Scala, there are also APIs which integrate SQL queries more closely such as Slick.[1] Its frontend allows to define and combine type-safe queries, similarly to ours (also in the way it is implemented). However, the language for defining queries maps to SQL, so it does not support nesting collections in other collections (a feature which simplified our example in Sec. 2.2), nor distinguishes statically between different kinds of collections, such as `Set` or `Seq`. Based on Ferry, ScalaQL [Garcia et al., 2010] extends Scala with a compiler-plugin to integrate a query language on top of a relational database. The work by Spiewak and Zhao [2009] is unrelated to [Garcia et al., 2010] but also called ScalaQL. It is similar to our approach in that it also proposes to reify queries based on for-comprehensions, but it is not clear from their paper how the reification works.[2]

8.2 Query Optimization

Query optimization on relational data is a long-standing issue in the database community, but there are also many works on query optimization on objects [Fegaras and Maier, 2000; Grust, 1999]. Compared to these works, we have only implemented a few simple query optimizations, so there is potential for further improvement of our work by incorporating more advanced optimizations.

Henglein [2010] and Henglein and Larsen [2010] embed relational algebra in Haskell. Queries are not reified in their approach, but due to a particularly sophisticated representation of multisets it is possible to execute some queries containing cross-products using faster equi-joins. While their approach appears to not rely on non-confluent rewriting rules and hence appears more robust, it is not yet clear how to incorporate other optimizations.

8.3 Deep Embeddings in Scala

Technically, our implementation of SQUOPT is a deep embedding of a part of the Scala collections API [Odersky and Moors, 2009]. Deep embeddings were pionereed by Leijen and Meijer [1999] and Elliott et al. [2003] in Haskell for other applications.

We regard the Scala collections API [Odersky and Moors, 2009] as a shallowly embedded query DSL. Query operators are *eager*, that is they immediately perform collection operations when called, so that it is not possible to optimize queries before execution.

Scala collections also provide *views*, which are closer to SQUOPT. Unlike standard query operators, they create *lazy* collections. Like SQUOPT, views reify query operators as data structures and interpret them later. Unlike SQUOPT, views are not used for automatic query optimization,

[1] http://slick.typesafe.com/

[2] We contacted the authors; they were not willing to provide more details or the sources of their approach.

but for explicitly changing the evaluation order of collection processing. Moreover, they cannot be reused by SQuOpt as they reify too little: Views embed deeply only the outermost pipeline of collection operators, while they embed shallowly nested collection operators and other Scala code in queries, such as arguments of `filter`, `map` and `flatMap`. Deep embedding of the whole query is necessary for many optimizations, as discussed in Chapter 3.

8.3.1 LMS

Our deep embedding is inspired by some of the Scala techniques presented by Lightweight Modular Staging (LMS) [Rompf and Odersky, 2010] for using implicits and for adding infix operators to a type. Like Rompf and Odersky [2010], we also generate and compile Scala code on-the-fly reusing the Scala compiler. A plausible alternative backend for SQuOpt would have been to use LMS and Delite [Rompf et al., 2011], a framework for building highly efficient DSLs in Scala.

An alternative to SQuOpt based on LMS and Delite, named OptiQL, was indeed built and presented in concurrent [Rompf et al., 2013] and subsequent work [Sujeeth et al., 2013b]. Like SQuOpt, OptiQL enables writing and optimizing collection queries in Scala. On the minus side, OptiQL supports fewer collections types (`ArrayList` and `HashMap`).

On the plus side, OptiQL supports queries containing effects and can reuse support for automatic parallelization and multiple platforms present in Delite. While LMS allows embedding effectful queries, it is unclear how many of the implemented optimizations keep being sound on such queries, and how many of those have been extended to such queries.

OptiQL achieves impressive speedups by fusing collection operations and transforming (or *lowering*) high-level bulk operators to highly optimized imperative programs using `while` loops. This gains significant advantages because it can avoid boxing, intermediate allocations, and because inlining heuristics in the Hotspot JIT compiler have never been tuned to bulk operators in functional programs.[3] We did not investigate such lowerings. It is unclear how well these optimizations would extend to other collections which intrinsically carry further overhead. Moreover, it is unclear how to execute incrementally the result of these lowerings, as we discuss in Sec. 7.1.

Those works did not support embedding arbitrary libraries in an automated or semi-automated way; this was only addressed later in Forge [Sujeeth et al., 2013a] and Yin-Yang [Jovanovic et al., 2014].

Ackermann et al. [2012] present Jet, which also optimizes collection queries. However, it targets MapReduce-style computations in a distributed environment. Like OptiQL, Jet does not apply typical database optimizations such as indexing or filter hoisting.

8.4 Code Querying

In our evaluation we explore the usage of SQuOpt to express queries on code and re-implement a subset of the FindBugs [Hovemeyer and Pugh, 2004] analyses. There are various other specialized code query languages such as CodeQuest [Hajiyev et al., 2006] or D-CUBED [Węgrzynowicz and Stencel, 2009]. Since these are special-purpose query languages that are not embedded into a host language, they are not directly comparable to our approach.

[3] See discussion by Cliff Click at http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem.

Chapter 9

Conclusions

We have illustrated the tradeoff between performance and modularity for queries on in-memory collections. We have shown that it is possible to design a deep embedding of a version of the collections API which reifies queries and can optimize them at runtime. Writing queries using this framework is, except minor syntactic details, the same as writing queries using the collection library, hence the adoption barrier to using our optimizer is low.

Our evaluation shows that using abstractions in queries introduces a significant performance overhead with native Scala code, while SQuOpt, in most cases, makes the overhead much more tolerable or removes it completely. Optimizations are not sufficient on some queries, but since our optimizer is a proof-of-concept with many opportunities for improvement, we believe a more elaborate version will achieve even better performance and reduce these limitations.

9.1 Future work

To make our DSL more convenient to use, it would be useful to use the virtualized pattern matcher of Scala 2.10, when it will be more robust, to add support for pattern matching in our virtualized queries.

Finally, while our optimizations are type-safe, as they rewrite an expression tree to another of the same type, currently the Scala type-checker cannot verify this statically, because of its limited support for GADTs. Solving this problem conveniently would allow checking statically that transformations are safe and make developing them easier.

Part II

Incremental λ-Calculus

Chapter 10

Introduction to differentiation

Incremental computation (or incrementalization) has a long-standing history in computer science [Ramalingam and Reps, 1993]. Often, a program needs to update quickly the output of some nontrivial function f when the input to the computation changes. In this scenario, we assume we have computed $y_1 = f\ x_1$ and we need to compute y_2 that equals $f\ x_2$. In this scenario, programmers typically have to choose between a few undesirable options.

- Programmers can call again function f on the updated input x_2 and repeat the computation from scratch. This choice guarantees correct results and is easy to implement, but typically wastes computation time. Often, if the updated input is close to the original input, the same result can be computed much faster.

- Programmers can write by hand a new function df that updates the output based on input changes, using various techniques. Running a hand-written function df can be much more efficient than rerunning f, but writing df requires significant developer effort, is error-prone, and requires updating df by hand to keep it consistent with f whenever f is modified. In practice, this complicates code maintenance significantly [Salvaneschi and Mezini, 2013].

- Programmers can write f using domain-specific languages that support incrementalization, for tasks where such languages are available. For instance, build scripts (our f) are written in domain-specific languages that support (coarse-grained) incremental builds. Database query languages also have often support for incrementalization.

- Programmers can attempt using general-purpose techniques for incrementalizing programs, such as *self-adjusting computation* and variants such as *Adapton*. Self-adjusting computation applies to arbitrary purely functional programs and has been extended to imperative programs; however, it only guarantees efficient incrementalization when applied to base programs that are *designed* for efficient incrementalization. Nevertheless, self-adjusting computation enabled incrementalizing programs that had never been incrementalized by hand before.

No approach guarantees automatic efficient incrementalization for arbitrary programs. We propose instead to design domain-specific languages (DSLs) that can be efficiently incrementalized, that we call *incremental* DSLs (IDSLs).

To incrementalize IDSL programs, we use a transformation that we call *(finite) differentiation*. Differentiation produces programs in the same language, called derivatives, that can be optimized further and compiled to efficient code. Derivatives represent changes to values through further values, that we call simply changes.

For primitives, IDSL designers must specify the result of differentiation: IDSL designers are to choose primitives that encapsulate efficiently incrementalizable computation schemes, while IDSL users are to express their computation using the primitives provided by the IDSL.

Helping IDSL designers to incrementalize primitives automatically is a desirable goal, though one that we leave open. In our setting, incrementalizing primitives becomes a problem of *program synthesis*, and we agree with Shah and Bodik [2017] that it should be treated as such. Among others, Liu [2000] develops a systematic approach to this synthesis problem for first-order programs based on equational reasoning, but it is unclear how scalable this approach is. We provide foundations for using equational reasoning, and sketch an IDSL for handling different sorts of collections. We also discuss avenues at providing language plugins for more fundamental primitives, such as algebraic datatypes with structural recursion.

In the IDSLs we consider, similarly to database languages, we use primitives for high-level operations, of complexity similar to SQL operators. On the one hand, IDSL designers wish to design few general primitives to limit the effort needed for manual incrementalization. On the other hand, overly general primitives can be harder to incrementalize efficiently. Nevertheless, we also provide some support for more general building blocks such as product or sum types and even (in some situations) recursive types. Other approaches provide more support for incrementalizing primitives, but even then ensuring efficient incrementalization is not necessarily easy. Dynamic approaches to incrementalization are most powerful: they can find work that can be reused at runtime using memoization, as long as the computation is structured so that memoization matches will occur. Moreover, for some programs it seems more efficient to detect that some output can be reused thanks to a description of the input changes, rather than through runtime detection.

We propose that IDSLs be higher-order, so that primitives can be parameterized over functions and hence highly flexible, and purely functional, to enable more powerful optimizations both before and after differentiation. Hence, an incremental DSL is a higher-order purely functional language, composed of a λ-calculus core extended with base types and primitives. Various database query languages support forms of finite differentiation (see Sec. 19.2.1), but only over first-order languages, which provide only restricted forms of operators such as *map*, *filter* or aggregation.

To support higher-order IDSLs, we define the first form of differentiation that supports higher-order functional languages; to this end, we introduce the concept of function changes, which contain changes to either the code of a function value or the values it closes over. While higher-order programs can be transformed to first-order ones, incrementalizing resulting programs is still beyond reach for previous approaches to differentiation (see Sec. 19.2.1 for earlier work and Sec. 19.2.2 for later approaches). In Chapter 17 and Appendix D we transform higher-order programs to first-order ones by closure conversion or defunctionalization, but we incrementalize defunctionalized programs using similar ideas, including changes to (defunctionalized) functions.

Our primary example will be DSLs for operations on collections: as discussed earlier (Chapter 2), we favor higher-order collection DSLs over relational databases.

We build our incremental DSLs based on simply-typed λ-calculus (STLC), extended with *language plugins* to define the domain-specific parts, as discussed in Appendix A.2 and summarized in Fig. A.1. We call our approach *ILC* for *Incremental Lambda Calculus*.

The rest of this chapter is organized as follows. In Sec. 10.1 we explain that differentiation generalizes the calculus of finite differences, a relative of differential calculus. In Sec. 10.2 we show a motivating example for our approach. In Sec. 10.3 we introduce informally the concept of *changes* as values, and in Sec. 10.4 we introduce *changes to functions*. In Sec. 10.5 we define differentiation and motivate it informally. In Sec. 10.6 we apply differentiation to our motivating example.

Correctness of ILC is far from obvious. In Chapters 12 and 13, we introduce a formal theory of changes, and we use it to formalize differentiation and prove it correct.

10.1 Generalizing the calculus of finite differences

Our theory of changes generalizes an existing field of mathematics called the *calculus of finite difference*: If f is a real function, one can define its *finite difference*, that is a function Δf such that $\Delta f\ a\ da = f\ (a + da) - f\ a$. Readers might notice the similarity (and the differences) between the finite difference and the derivative of f, since the latter is defined as

$$f'(a) = \lim_{da \to 0} \frac{f(a + da) - f(a)}{da}.$$

The calculus of finite differences helps computing a closed formula for Δf given a closed formula for f. For instance, if function f is defined by $f\ x = 2 \cdot x$, one can prove its finite difference is $\Delta f\ x\ dx = 2 \cdot (x + dx) - 2 \cdot x = 2 \cdot dx$.

Finite differences are helpful for incrementalization because they allow computing functions on updated inputs based on results on base inputs, if we know how inputs change. Take again for instance $f\ x = 2 \cdot x$: if x is a base input and $x + dx$ is an updated input, we can compute $f\ (x + dx) = f\ x + \Delta f\ x\ dx$. If we already computed $y = f\ x$ and reuse the result, we can compute $f\ (x + dx) = y + \Delta f\ x$. Here, the input change is dx and the output change is $\Delta f\ x\ dx$.

However, the calculus of finite differences is usually defined for real functions. Since it is based on operators + and −, it can be directly extended to commutative groups. Incrementalization based on finite differences for groups and first-order programs has already been researched [Paige and Koenig, 1982; Gluche et al., 1997], most recently and spectacularly with DBToaster [Koch, 2010; Koch et al., 2016].

But it is not immediate how to generalize finite differencing beyond groups. And many useful types do not form a group: for instance, lists of integers don't form a group but only a monoid. Moreover, it's hard to represent list changes simply through a list: how do we specify which elements were inserted (and where), which were removed and which were subjected to change themselves?

In ILC, we generalize the calculus of finite differences by using distinct types for base values and changes, and adapting the surrounding theory. ILC generalizes operators + and − as operators $\oplus$ (pronounced "oplus" or "update") and $\ominus$ (pronounced "ominus" or "difference"). We show how ILC subsumes groups in Sec. 13.1.1.

10.2 A motivating example

In this section, we illustrate informally incrementalization on a small example.

In the following program, *grandTotal xs ys* sums integer numbers in input collections *xs* and *ys*.

$$\begin{aligned}
&\mathit{grandTotal} && :: \mathit{Bag}\ \mathbb{Z} \to \mathit{Bag}\ \mathbb{Z} \to \mathbb{Z} \\
&s && :: \mathbb{Z} \\
&\mathit{grandTotal}\ \mathit{xs}\ \mathit{ys} &&= \mathit{sum}\ (\mathit{merge}\ \mathit{xs}\ \mathit{ys}) \\
&s && = \mathit{grandTotal}\ \mathit{xs}\ \mathit{ys}
\end{aligned}$$

This program computes output *s* from input collections *xs* and *ys*. These collections are multisets or *bags*, that is, collections that are unordered (like sets) where elements are allowed to appear more than once (unlike sets). In this example, we assume a language plugin that supports a base type of integers $\mathbb{Z}$ and a family of base types of bags $\mathit{Bag}\ \tau$ for any type τ.

We can run this program on specific inputs $xs_1 = \{\{1, 2, 3\}\}$ and $ys_1 = \{\{4\}\}$ to obtain output s_1. Here, double braces $\{\{\ldots\}\}$ denote a bag containing the elements among the double braces.

$$\begin{aligned}
s_1 &= \mathit{grandTotal}\ xs_1\ ys_1 \\
&= \mathit{sum}\ \{\{1, 2, 3, 4\}\} = 10
\end{aligned}$$

This example uses small inputs for simplicity, but in practice they are typically much bigger; we use n to denote the input size. In this case the asymptotic complexity of computing s is $\Theta(n)$.

Consider computing updated output s_2 from updated inputs $xs_2 = \{\{1, 1, 2, 3\}\}$ and $ys_2 = \{\{4, 5\}\}$. We could recompute s_2 from scratch as

$$\begin{aligned} s_2 &= grandTotal\ xs_2\ ys_2 \\ &= sum\ \{\{1, 1, 2, 3, 4, 5\}\} = 16 \end{aligned}$$

But if the size of the updated inputs is $\Theta(n)$, recomputation also takes time $\Theta(n)$, and we would like to obtain our result asymptotically faster.

To compute the updated output s_2 faster, we assume the changes to the inputs have a description of size dn that is asymptotically smaller than the input size n, that is $dn = o(n)$. All approaches to incrementalization require small input changes. Incremental computation will then process the input changes, rather than just the new inputs.

10.3 Introducing changes

To talk about how the differences between old values and new values, we introduce a few concepts, for now without full definitions. In our approach to incrementalization, we describe changes to values as values themselves: We call such descriptions simply *changes*. Incremental programs examine changes to inputs to understand how to produce changes to outputs. Just like in STLC we have terms (programs) that evaluates to values, we also have *change terms*, which evaluate to *change values*. We require that going from old values to new values preserves types: That is, if an old value v_1 has type τ, then also its corresponding new value v_2 must have type τ. To each type τ we associate a type of changes or *change type* $\Delta\tau$: a change between v_1 and v_2 must be a value of type $\Delta\tau$. Similarly, environments can change: to typing context Γ we associate change typing contexts $\Delta\Gamma$, such that we can have an environment change $d\rho : [\![\ \Delta\Gamma\ ]\!]$ from $\rho_1 : [\![\ \Gamma\ ]\!]$ to $\rho_2 : [\![\ \Gamma\ ]\!]$.

Not all descriptions of changes are meaningful, so we also talk about *valid* changes. Valid changes satisfy additional invariants that are useful during incrementalization. A change value dv can be a valid change from v_1 to v_2. We can also consider a valid change as an edge from v_1 to v_2 in a graph associated to τ (where the vertexes are values of type τ), and we call v_1 the source of dv and v_2 the destination of dv. We only talk of source and destination for valid changes: so a change from v_1 to v_2 is (implicitly) valid. We'll discuss examples of valid and invalid changes in Examples 10.3.1 and 10.3.2.

We also introduce an operator $\oplus$ on values and changes: if dv is a valid change from v_1 to v_2, then $v_1 \oplus dv$ (read as "v_1 updated by dv") is guaranteed to return v_2. If dv is *not* a valid change from v_1, then $v_1 \oplus dv$ can be defined to some arbitrary value or not, without any effect on correctness. In practice, if $\oplus$ detects an invalid input change it can trigger an error or return a dummy value; in our formalization we assume for simplicity that $\oplus$ is total. Again, if dv is not valid from v_1 to $v_1 \oplus dv$, then we do not talk of the source and destination of dv.

We also introduce operator $\ominus$: given two values v_1, v_2 for the same type, $v_2 \ominus v_1$ is a valid change from v_1 to v_2.

Finally, we introduce change composition: if dv_1 is a valid change from v_1 to v_2 and dv_2 is a valid change from v_2 to v_3, then $dv_1 \odot dv_2$ is a valid change from v_1 to v_3.

Change operators are overloaded over different types. Coherent definitions of validity and of operators $\oplus$, $\ominus$ and $\odot$ for a type τ form a *change structure* over values of type τ (Definition 13.1.1). For each type τ we'll define a change structure (Definition 13.4.2), and operators will have types $\oplus : \tau \to \Delta\tau \to \tau$, $\ominus : \tau \to \tau \to \Delta\tau$, $\odot : \Delta\tau \to \Delta\tau \to \Delta\tau$.

Example 10.3.1 (Changes on integers and bags)
To show how incrementalization affects our example, we next describe valid changes for integers and bags. For now, a change das to a bag as_1 simply contains all elements to be added to the initial bag as_1 to obtain the updated bag as_2 (we'll ignore removing elements for this section and discuss it later). In our example, the change from xs_1 (that is $\{\{1,2,3\}\}$) to xs_2 (that is $\{\{1,1,2,3\}\}$) is $dxs = \{\{1\}\}$, while the change from ys_1 (that is $\{\{4\}\}$) to ys_2 (that is $\{\{4,5\}\}$) is $dys = \{\{5\}\}$. To represent the output change ds from s_1 to s_2 we need integer changes. For now, we represent integer changes as integers, and define $\oplus$ on integers as addition: $v_1 \oplus dv = v_1 + dv$. □

For both bags and integers, a change dv is always valid between v_1 and $v_2 = v_1 \oplus dv$; for other changes, however, validity will be more restrictive.

Example 10.3.2 (Changes on naturals)
For instance, say we want to define changes on a type of natural numbers, and we still want to have $v_1 \oplus dv = v_1 + dv$. A change from 3 to 2 should still be -1, so the type of changes must be $\mathbb{Z}$. But the result of $\oplus$ should still be a natural, that is an integer $\geqslant 0$: to ensure that $v_1 \oplus dv \geqslant 0$ we need to require that $dv \geqslant -v_1$. We use this requirement to define validity on naturals: $dv \triangleright v_1 \hookrightarrow v_1 + dv : \mathbb{N}$ is defined as equivalent to $dv \geqslant -v_1$. We can guarantee equation $v_1 \oplus dv = v_1 + dv$ not for all changes, but only for valid changes. Conversely, if a change dv is invalid for v_1, then $v_1 + dv < 0$. We then define $v_1 \oplus dv$ to be 0, though any other definition on invalid changes would work.[1] □

10.3.1 Incrementalizing with changes

After introducing changes and related notions, we describe how we incrementalize our example program.

We consider again the scenario of Sec. 10.2: we need to compute the updated output s_2, the result of calling our program *grandTotal* on updated inputs xs_2 and ys_2. And we have the initial output s_1 from calling our program on initial inputs xs_1 and ys_1. In this scenario we can compute s_2 *non-incrementally* by calling *grandTotal* on the updated inputs, but we would like to obtain the same result faster. Hence, we compute s_2 *incrementally*: that is, we first compute the *output change* ds from s_1 to s_2; then we update the old output s_1 by change ds. Successful incremental computation must compute the correct s_2 asymptotically faster than non-incremental computation. This speedup is possible because we take advantage of the computation already done to compute s_1.

To compute the output change ds from s_1 to s_2, we propose to transform our *base program* *grandTotal* to a new program *dgrandTotal*, that we call the *derivative* of *grandTotal*: to compute ds we call *dgrandTotal* on initial inputs and their respective changes. Unlike other approaches to incrementalization, *dgrandTotal* is a regular program in the same language as *grandTotal*, hence can be further optimized with existing technology.

Below, we give the code for *dgrandTotal* and show that in this example incremental computation computes s_2 correctly.

For ease of reference, we recall inputs, changes and outputs:

$$
\begin{aligned}
xs_1 &= \{\{1,2,3\}\} \\
dxs &= \{\{1\}\} \\
xs_2 &= \{\{1,1,2,3\}\} \\
ys_1 &= \{\{4\}\} \\
dys &= \{\{5\}\}
\end{aligned}
$$

[1] In fact, we could leave $\oplus$ undefined on invalid changes. Our original presentation [Cai et al., 2014], in essence, restricted $\oplus$ to valid changes through dependent types, by ensuring that applying it to invalid changes would be ill-typed. Later, Huesca [2015], in similar developments, simply made $\oplus$ partial on its domain instead of restricting the domain, achieving similar results.

$$
\begin{aligned}
ys_2 &= \{\{4,5\}\} \\
s_1 &= grandTotal\ xs_1\ ys_1 \\
&= 10 \\
s_2 &= grandTotal\ xs_2\ ys_2 \\
&= 16
\end{aligned}
$$

Incremental computation uses the following definitions to compute s_2 correctly and with fewer steps, as desired.

$$
\begin{aligned}
&dgrandTotal\ xs\ dxs\ ys\ dys = sum\ (merge\ dxs\ dys) \\
&ds = dgrandTotal\ xs_1\ dxs\ ys_1\ dys = \\
&\quad = sum\ \{\{1,5\}\} = 6 \\
&s_2 = s_1 \oplus ds = s_1 + ds \\
&\quad = 10 + 6 = 16
\end{aligned}
$$

Incremental computation should be asymptotically faster than non-incremental computation; hence, the derivative we run should be asymptotically faster than the base program. Here, derivative *dgrandTotal* is faster simply because it *ignores* initial inputs altogether. Therefore, its time complexity depends only on the total size of changes *dn*. In particular, the complexity of *dgrandTotal* is $\Theta(dn) = o(n)$.

We generate derivatives through a program transformation from terms to terms, which we call *differentiation* (or, sometimes, simply *derivation*). We write $\mathcal{D}[\![\ t\ ]\!]$ for the result of differentiating term *t*. We apply $\mathcal{D}[\![\ -\ ]\!]$ on terms of our non-incremental programs or *base terms*, such as *grandTotal*. To define differentiation, we assume that we already have derivatives for primitive functions they use; we discuss later how to write such derivatives by hand.

We define differentiation in Definition 10.5.1; some readers might prefer to peek ahead, but we prefer to first explain what differentiation is supposed to do.

A derivative of a function can be applied to initial inputs and changes from initial inputs to updated inputs, and returns a change from an initial output to an updated output. For instance, take derivative *dgrandTotal*, initial inputs xs_1 and ys_1, and changes *dxs* and *dys* from initial inputs to updated inputs. Then, change $dgrandTotal\ xs_1\ dxs\ ys_1\ dys$, that is *ds*, goes from initial output $grandTotal\ xs_1\ ys_1$, that is s_1, to updated output $grandTotal\ xs_2\ ys_2$, that is s_2. And because *ds* goes from s_1 to s_2, it follows as a corollary that $s_2 = s_1 \oplus ds$. Hence, we can compute s_2 incrementally through $s_1 \oplus ds$, as we have shown, rather than by evaluating $grandTotal\ xs_2\ ys_2$.

We often just say that a derivative of function f maps changes to the inputs of f to changes to the outputs of f, leaving the initial inputs implicit. In short:

Slogan 10.3.3
Term $\mathcal{D}[\![\ t\ ]\!]$ *maps input changes to output changes.* That is, $\mathcal{D}[\![\ t\ ]\!]$ applied to initial base inputs and valid *input changes* (from initial inputs to updated inputs) gives a valid *output change* from *t* applied on old inputs to *t* applied on new inputs. □

For a generic unary function $f : A \to B$, the behavior of $\mathcal{D}[\![\ f\ ]\!]$ can be described as:

$$f\ a_2 \cong f\ a_1 \oplus \mathcal{D}[\![\ f\ ]\!]\ a_1\ da \tag{10.1}$$

or as

$$f\ (a_1 \oplus da) \cong f\ a_1 \oplus \mathcal{D}[\![\ f\ ]\!]\ a_1\ da \tag{10.2}$$

where *da* is a metavariable standing for a valid change from a_1 to a_2 (with $a_1, a_2 : A$) and where $\cong$ denotes denotational equivalence (Definition A.2.5). Moreover, $\mathcal{D}[\![\ f\ ]\!]\ a_1\ da$ is also a valid change and can be hence used as an argument for operations that require valid changes. These equations

follow from Theorem 12.2.2 and Corollary 13.4.5; we iron out the few remaining details to obtain these equations in Sec. 14.1.[2]

In our example, we have applied $\mathcal{D}[\![-]\!]$ to *grandTotal*, and simplify the result via β-reduction to produce *dgrandTotal*, as we show in Sec. 10.6. Correctness of $\mathcal{D}[\![-]\!]$ guarantees that *sum* (*merge dxs dys*) evaluates to a change from *sum* (*merge xs ys*) evaluated on old inputs xs_1 and ys_1 to *sum* (*merge xs ys*) evaluated on new inputs xs_2 and ys_2.

In this section, we have sketched the meaning of differentiation informally. We discuss incrementalization on higher-order terms in Sec. 10.4, and actually define differentiation in Sec. 10.5.

10.4 Differentiation on open terms and functions

We have shown that applying $\mathcal{D}[\![-]\!]$ on closed functions produces their derivatives. However, $\mathcal{D}[\![-]\!]$ is defined for all terms, hence also for open terms and for non-function types.

Open terms $\Gamma \vdash t : \tau$ are evaluated with respect to an environment for Γ, and when this environment changes, the result of t changes as well; $\mathcal{D}[\![t]\!]$ computes the change to t's output. If $\Gamma \vdash t : \tau$, evaluating term $\mathcal{D}[\![t]\!]$ requires as input a *change environment* $d\rho : [\![\Delta\Gamma]\!]$ containing changes from the *initial environment* $\rho_1 : [\![\Gamma]\!]$ to the *updated environment* $\rho_2 : [\![\Gamma]\!]$. The (environment) input change $d\rho$ is mapped by $\mathcal{D}[\![t]\!]$ to output change $dv = [\![\mathcal{D}[\![t]\!]]\!]\ d\rho$, a change from *initial output* $[\![t]\!]\ \rho_1$ to *updated output* $[\![t]\!]\ \rho_2$. If t is a function, dv maps in turn changes to the function arguments to changes to the function result. All this behavior, again, follows our slogan.

Environment changes contains changes for each variable in Γ. More precisely, if variable x appears with type τ in Γ and hence in ρ_1, ρ_2, then dx appears with type $\Delta\tau$ in $\Delta\Gamma$ and hence in $d\rho$. Moreover, $d\rho$ extends ρ_1 to provide $\mathcal{D}[\![t]\!]$ with initial inputs and not just with their changes.

The two environments ρ_1 and ρ_2 can share entries — in particular, environment change $d\rho$ can be a *nil change* from ρ to ρ. For instance, Γ can be empty: then ρ_1 and ρ_2 are also empty (since they match Γ) and equal, so $d\rho$ is a nil change. Alternatively, some or all the change entries in $d\rho$ can be nil changes. Whenever $d\rho$ is a nil change, $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ is also a nil change.

If t is a function, $\mathcal{D}[\![t]\!]$ will be a *function change*. Changes to functions in turn map input changes to output changes, following our Slogan 10.3.3. If a change df from f_1 to f_2 is applied (via $df\ a_1\ da$) to an input change da from a_1 to a_2, then df will produce a change $dv = df\ a_1\ da$ from $v_1 = f_1\ a_1$ to $v_2 = f_2\ a_2$. The definition of function changes is recursive on types: that is, dv can in turn be a function change mapping input changes to output changes.

Derivatives are a special case of function changes: a derivative df is simply a change from f to f itself, which maps input changes da from a_1 to a_2 to output changes $dv = df\ a_1\ da$ from $f\ a_1$ to $f\ a_2$. This definition coincides with the earlier definition of derivatives, and it also coincides with the definition of function changes for the special case where $f_1 = f_2 = f$. That is why $\mathcal{D}[\![t]\!]$ produces derivatives if t is a closed function term: we can only evaluate $\mathcal{D}[\![t]\!]$ against an nil environment change, producing a nil function change.

Since the concept of function changes can be surprising, we examine it more closely next.

10.4.1 Producing function changes

A first-class function can close over free variables that can change, hence functions values themselves can change; hence, we introduce *function changes* to describe these changes.

For instance, term $t_f = \lambda x \to x + y$ is a function that closes over y, so different values v for y give rise to different values for $f = [\![t_f]\!]\ (y = v)$. Take a change dv from $v_1 = 5$ to $v_2 = 6$; different

[2]Nitpick: if *da* is read as an object variable, denotational equivalence will detect that these terms are not equivalent if *da* maps to an invalid change. Hence we said that *da* is a metavariable. Later we define denotational equivalence for valid changes (Definition 14.1.2), which gives a less cumbersome way to state such equations.

inputs v_1 and v_2 for y give rise to different outputs $f_1 = [\![t_f]\!]\ (y = v_1)$ and $f_2 = [\![t_f]\!]\ (y = v_2)$. We describe the difference between outputs f_1 and f_2 through a function change df from f_1 to f_2.

Consider again Slogan 10.3.3 and how it applies to term f:

Slogan 10.3.3
Term $\mathcal{D}[\![t]\!]$ *maps input changes to output changes.* That is, $\mathcal{D}[\![t]\!]$ applied to initial base inputs and valid *input changes* (from initial inputs to updated inputs) gives a valid *output change* from t applied on old inputs to t applied on new inputs. □

Since y is free in t_f, the value for y is an input of t_f. So, continuing our example, $dt_f = \mathcal{D}[\![t_f]\!]$ must map a valid input change dv from v_1 to v_2 for variable y to a valid output change df from f_1 to f_2; more precisely, we must have $df = [\![dt_f]\!]\ (y = v_1, dy = dv)$.

10.4.2 Consuming function changes

Function changes can not only be produced but also be consumed in programs obtained from $\mathcal{D}[\![-]\!]$. We discuss next how.

As discussed, we consider the value for y as an input to $t_f = \lambda x \to x + y$. However, we also choose to consider the argument for x as an input (of a different sort) to $t_f = \lambda x \to x + y$, and we require our Slogan 10.3.3 to apply to input x too. While this might sound surprising, it works out well. Specifically, since $df = [\![\mathcal{D}[\![t_f]\!]]\!]$ is a change from f_1 to f_2, we require $df\ a_1\ da$ to be a change from $f_1\ a_1$ to $f_2\ a_2$, so df maps base input a_1 and input change da to output change $df\ a_1\ da$, satisfying the slogan.

More in general, any valid function change df from f_1 to f_2 (where $f_1, f_2 : [\![\sigma \to \tau]\!]$) must in turn be a function that takes an input a_1 and a change da, valid from a_1 to a_2, to a valid change $df\ a_1\ da$ from $f_1\ a_1$ to $f_2\ a_2$.

This way, to satisfy our slogan on application $t = t_f\ x$, we can simply define $\mathcal{D}[\![-]\!]$ so that $\mathcal{D}[\![t_f\ x]\!] = \mathcal{D}[\![t_f]\!]\ x\ dx$. Then

$$[\![\mathcal{D}[\![t_f\ x]\!]]\!]\ (y = v_1, dy = dv, x = a_1, dx = da) = [\![\mathcal{D}[\![t_f]\!]]\!]\ a_1\ da = df\ a_1\ da.$$

As required, that's a change from $f_1\ a_1$ to $f_2\ a_2$.

Overall, valid function changes preserve validity, just like $\mathcal{D}[\![t]\!]$ in Slogan 10.3.3, and map valid input changes to valid output changes. In turn, output changes can be function changes; since they are valid, they in turn map valid changes to their inputs to valid output changes (as we'll see in Lemma 12.1.10). We'll later formalize this and define validity by recursion on types, that is, as a *logical relation* (see Sec. 12.1.4).

10.4.3 Pointwise function changes

It might seem more natural to describe a function change df' between f_1 and f_2 via $df'\ x = \lambda x \to f_2\ x \ominus f_1\ x$. We call such a df' a pointwise change. While some might find pointwise changes a more natural concept, the output of differentiation needs often to compute $f_2\ x_2 \ominus f_1\ x_1$, using a change df from f_1 to f_2 and a change dx from x_1 to x_2. Function changes perform this job directly. We discuss this point further in Sec. 15.3.

10.4.4 Passing change targets

It would be more symmetric to make function changes also take updated input a_2, that is, have $df\ a_1\ da\ a_2$ computes a change from $f_1\ a_1$ to $f_2\ a_2$. However, passing a_2 explicitly adds no information: the value a_2 can be computed from a_1 and da as $a_1 \oplus da$. Indeed, in various cases

a function change can compute its required output without actually computing $a_1 \oplus da$. Since we expect the size of a_1 and a_2 is asymptotically larger than da, actually computing a_2 could be expensive.[3] Hence, we stick to our asymmetric form of function changes.

10.5 Differentiation, informally

Next, we define differentiation and explain informally why it does what we want. We then give an example of how differentiation applies to our example. A formal proof will follow soon in Sec. 12.2, justifying more formally why this definition is correct, but we proceed more gently.

Definition 10.5.1 (Differentiation)
Differentiation is the following term transformation:

$$\begin{aligned}
\mathcal{D}[\![\, \lambda(x : \sigma) \to t \,]\!] &= \lambda(x : \sigma)\ (dx : \Delta\sigma) \to \mathcal{D}[\![\, t \,]\!] \\
\mathcal{D}[\![\, s\ t \,]\!] &= \mathcal{D}[\![\, s \,]\!]\ \ t\ \mathcal{D}[\![\, t \,]\!] \\
\mathcal{D}[\![\, x \,]\!] &= dx \\
\mathcal{D}[\![\, c \,]\!] &= \mathcal{D}^C[\![\, c \,]\!]
\end{aligned}$$

□

where $\mathcal{D}^C[\![\, c \,]\!]$ defines differentiation on primitives and is provided by language plugins (see Appendix A.2.5), and dx stands for a variable generated by prefixing x's name with d, so that $\mathcal{D}[\![\, y \,]\!] = dy$ and so on.

If we extend the language with (non-recursive) **let**-bindings, we can give derived rules for it such as:

$$\begin{aligned}
\mathcal{D}[\![\, \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \,]\!] = \textbf{let}\ & x = t_1 \\
& dx = \mathcal{D}[\![\, t_1 \,]\!] \\
\textbf{in}\ & \mathcal{D}[\![\, t_2 \,]\!]
\end{aligned}$$

In Sec. 15.1 we will explain that the same transformation rules apply for *recursive* **let**-bindings.

If t contains occurrences of both (say) x and dx, capture issues arise in $\mathcal{D}[\![\, t \,]\!]$. We defer these issues to Sec. 12.3.4, and assume throughout that such issues can be avoided by α-renaming and the Barendregt convention [Barendregt, 1984].

This transformation might seem deceptively simple. Indeed, pure λ-calculus only handles binding and higher-order functions, leaving "real work" to primitives. Similarly, our transformation incrementalizes binding and higher-order functions, leaving "real work" to derivatives of primitives. However, our support of λ-calculus allows to *glue* primitives together. We'll discuss later how we add support to various primitives and families of primitives.

Now we try to motivate the transformation informally. We claimed that $\mathcal{D}[\![\, - \,]\!]$ must satisfy Slogan 10.3.3, which reads

Slogan 10.3.3
Term $\mathcal{D}[\![\, t \,]\!]$ maps input changes to output changes. That is, $\mathcal{D}[\![\, t \,]\!]$ applied to initial base inputs and valid *input changes* (from initial inputs to updated inputs) gives a valid *output change* from t applied on old inputs to t applied on new inputs. □

Let's analyze the definition of $\mathcal{D}[\![\, - \,]\!]$ by case analysis of input term u. In each case we assume that our slogan applies to any subterms of u, and sketch why it applies to u itself.

[3] We show later efficient change structures where $\oplus$ reuses part of a_1 to output a_2 in logarithmic time.

- if $u = x$, by our slogan $\mathcal{D}[\![x]\!]$ must evaluate to the change of x when inputs change, so we set $\mathcal{D}[\![x]\!] = dx$.
- if $u = c$, we simply delegate differentiation to $\mathcal{D}^C[\![c]\!]$, which is defined by plugins. Since plugins can define arbitrary primitives, they need provide their derivatives.
- if $u = \lambda x \to t$, then u introduces a function. Assume for simplicity that u is a closed term. Then $\mathcal{D}[\![t]\!]$ evaluates to the change of the result of this function u, evaluated in a context binding x and its change dx. Then, because of how function changes are defined, the change of u is the change of output t as a function of the *base input* x and its change dx, that is $\mathcal{D}[\![u]\!] = \lambda x\ dx \to \mathcal{D}[\![t]\!]$.
- if $u = s\ t$, then s is a function. Assume for simplicity that u is a closed term. Then $\mathcal{D}[\![s]\!]$ evaluates to the change of s, as a function of $\mathcal{D}[\![s]\!]$'s base input and input change. So, we apply $\mathcal{D}[\![s]\!]$ to its actual base input t and actual input change $\mathcal{D}[\![t]\!]$, and obtain $\mathcal{D}[\![s\ t]\!] = \mathcal{D}[\![s]\!]\ t\ \mathcal{D}[\![t]\!]$.

This is not quite a correct proof sketch because of many issues, but we fix these issues with our formal treatment in Sec. 12.2. In particular, in the case for abstraction $u = \lambda x \to t$, $\mathcal{D}[\![t]\!]$ depends not only on x and dx, but also on other free variables of u and their changes. Similarly, we must deal with free variables also in the case for application $u = s\ t$. But first, we apply differentiation to our earlier example.

10.6 Differentiation on our example

To exemplify the behavior of differentiation concretely, and help fix ideas for later discussion, in this section we show how the derivative of *grandTotal* looks like.

$$\begin{aligned} &grandTotal = \lambda xs\ ys \to sum\ (merge\ xs\ ys) \\ &s \qquad\qquad = grandTotal\ \{\{1\}\}\ \{\{2,3,4\}\} = 11 \end{aligned}$$

Differentiation is a structurally recursive program transformation, so we first differentiate *merge xs ys*. To compute its change we simply call the derivative of *merge*, that is *dmerge*, and apply it to the base inputs and their changes: hence we write

$$\mathcal{D}[\![merge\ xs\ ys]\!] = dmerge\ xs\ dxs\ ys\ dys$$

As we'll better see later, we can define function *dmerge* as

$$dmerge = \lambda xs\ dxs\ ys\ dys \to merge\ dxs\ dys$$

so $\mathcal{D}[\![merge\ xs\ ys]\!]$ can be simplified by β-reduction to *merge dxs dys*:

$$\begin{aligned} &\mathcal{D}[\![merge\ xs\ ys]\!] \\ =\ &dmerge\ xs\ dxs\ ys\ dys \\ =_\beta\ &(\lambda xs\ dxs\ ys\ dys \to merge\ dxs\ dys)\ xs\ dxs\ ys\ dys \\ =_\beta\ &merge\ dxs\ dys \end{aligned}$$

Let's next derive *sum* (*merge xs ys*). First, like above, the derivative of *sum zs* would be *dsum zs dzs*, which depends on base input *zs* and its change *dzs*. As we'll see, *dsum zs dzs* can simply call *sum* on *dzs*, so *dsum zs dzs* = *sum dzs*. To derive *sum* (*merge xs ys*), we must call

the derivative of *sum*, that is *dsum*, on its base argument and its change, so on *merge xs ys* and $\mathcal{D}[\![\ merge\ xs\ ys\ ]\!]$. We can later simplify again by β-reduction and obtain

$$\begin{aligned}
&\mathcal{D}[\![\ sum\ (merge\ xs\ ys)\ ]\!] \\
=\ &dsum\ (merge\ xs\ ys)\ \mathcal{D}[\![\ merge\ xs\ ys\ ]\!] \\
=_\beta\ &sum\ \mathcal{D}[\![\ merge\ xs\ ys\ ]\!] \\
=\ &sum\ (dmerge\ xs\ dxs\ ys\ dys) \\
=_\beta\ &sum\ (merge\ dxs\ dys)
\end{aligned}$$

Here we see the output of differentiation is defined in a bigger typing context: while *merge xs ys* only depends on base inputs *xs* and *ys*, $\mathcal{D}[\![\ merge\ xs\ ys\ ]\!]$ also depends on their changes. This property extends beyond the examples we just saw: if a term t is defined in context Γ, then the output of derivation $\mathcal{D}[\![\ t\ ]\!]$ is defined in context $\Gamma, \Delta\Gamma$, where $\Delta\Gamma$ is a context that binds a change dx for each base input x bound in the context Γ.

Next we consider $\lambda xs\ ys \to sum\ (merge\ xs\ ys)$. Since variables *xs*, *dxs*, *ys*, *dys* appear free in $\mathcal{D}[\![\ sum\ (merge\ xs\ ys)\ ]\!]$ (ignoring later optimizations), term

$$\mathcal{D}[\![\ \lambda xs\ ys \to sum\ (merge\ xs\ ys)\ ]\!]$$

must bind all those variables.

$$\begin{aligned}
&\mathcal{D}[\![\ \lambda xs\ ys \to sum\ (merge\ xs\ ys)\ ]\!] \\
=\ &\lambda xs\ dxs\ ys\ dys \to \mathcal{D}[\![\ sum\ (merge\ xs\ ys)\ ]\!] \\
=_\beta\ &\lambda xs\ dxs\ ys\ dys \to sum\ (merge\ dxs\ dys)
\end{aligned}$$

Next we need to transform the binding of *grandTotal* to its body $b = \lambda xs\ ys \to sum\ (merge\ xs\ ys)$. We copy this binding and add a new additional binding from *dgrandTotal* to the derivative of b.

$$\begin{aligned}
grandTotal\ &= \lambda xs\ \ \ ys\ \ \ \to sum\ (merge\ xs\ \ ys) \\
dgrandTotal\ &= \lambda xs\ dxs\ ys\ dys \to sum\ (merge\ dxs\ dys)
\end{aligned}$$

Finally, we need to transform the binding of *output* and its body. By iterating similar steps, in the end we get:

$$\begin{aligned}
grandTotal\ &= \lambda xs\ \ \ ys\ \ \ \to sum\ (merge\ xs\ \ ys) \\
dgrandTotal\ &= \lambda xs\ dxs\ ys\ dys \to sum\ (merge\ dxs\ dys) \\
s\ &= grandTotal\ \ \{\{1,2,3\}\}\ \{\{4\}\} \\
ds\ &= dgrandTotal\ \{\{1,2,3\}\}\ \{\{1\}\}\ \{\{4\}\}\ \{\{5\}\} \\
&=_\beta sum\ (merge\ \{\{1\}\}\ \{\{5\}\})
\end{aligned}$$

Self-maintainability Differentiation does not always produce efficient derivatives without further program transformations; in particular, derivatives might need to recompute results produced by the base program. In the above example, if we don't inline derivatives and use β-reduction to simplify programs, $\mathcal{D}[\![\ sum\ (merge\ xs\ ys)\ ]\!]$ is just $dsum\ (merge\ xs\ ys)\ \mathcal{D}[\![\ merge\ xs\ ys\ ]\!]$. A direct execution of this program will compute *merge xs ys*, which would waste time linear in the base inputs.

We'll show how to avoid such recomputations in general in Sec. 17.2; but here we can avoid computing *merge xs ys* simply because *dsum* does not use its base argument, that is, it is *self-maintainable*. Without the approach described in Chapter 17, we are restricted to self-maintainable derivatives.

10.7 Conclusion and contributions

In this chapter, we have seen how a correct differentiation transform allows us to incrementalize programs.

10.7.1 Navigating this thesis part

Differentiation This chapter and Chapters 11 to 14 form the core of incrementalization theory, with other chapters building on top of them. We study incrementalization for STLC; we summarize our formalization of STLC in Appendix A. Together with Chapter 10, Chapter 11 introduces the overall approach to incrementalization informally. Chapters 12 and 13 show that incrementalization using ILC is correct. Building on a set-theoretic semantics, these chapters also develop the theory underlying this correctness proofs and further results. Equational reasoning on terms is then developed in Chapter 14.

Chapters 12 to 14 contain full formal proofs: readers are welcome to skip or skim those proofs where appropriate. For ease of reference and to help navigation and skimming, these highlight and number all definitions, notations, theorem statements and so on, and we strive not to hide important definitions inside theorems.

Later chapters build on this core but are again independent from each other.

Extensions and theoretical discussion Chapter 15 discusses a few assorted aspects of the theory that do not fit elsewhere and do not suffice for standalone chapters. We show how to differentiation general recursion Sec. 15.1, we exhibit a function change that is not valid for any function (Sec. 15.2), we contrast our representation of function changes with *pointwise* function changes (Sec. 15.3), and we compare our formalization with the one presented in [Cai et al., 2014] (Sec. 15.4).

Performance of differentiated programs Chapter 16 studies how to apply differentiation (as introduced in previous chapters) to incrementalize a case study and empirically evaluates performance speedups.

Differentiation with cache-transfer-style conversion Chapter 17 is self-contained, even though it builds on the rest of the material; it summarizes the basics of ILC in Sec. 17.2.1. Terminology in that chapter is sometimes subtly different from the rest of the thesis.

Towards differentiation for System F Chapter 18 outlines how to extend differentiation to System F and suggests a proof avenue. Differentiation for System F requires a generalization of change structures to changes across different types (Sec. 18.4) that appears of independent interest, though further research remains to be done.

Related work, conclusions and appendixes Finally, Sec. 17.6 and Chapter 19 discuss related work and Chapter 20 concludes this part of the thesis.

10.7.2 Contributions

In Chapters 11 to 14 and Chapter 16 we make the following contributions:

- We present a novel mathematical theory of changes and derivatives, which is more general than other work in the field because changes are first-class entities, they are distinct from base values and they are defined also for functions.

- We present the first approach to incremental computation for pure λ-calculi by a source-to-source transformation, $\mathcal{D}$, that requires no run-time support. The transformation produces an incremental program in the same language; all optimization techniques for the original program are applicable to the incremental program as well.
- We prove that our incrementalizing transformation $\mathcal{D}$ is correct by a novel machine-checked logical relation proof, mechanized in Agda.
- While we focus mainly on the theory of changes and derivatives, we also perform a performance case study. We implement the derivation transformation in Scala, with a plug-in architecture that can be extended with new base types and primitives. We define a plugin with support for different collection types and use the plugin to incrementalize a variant of the MapReduce programming model [Lämmel, 2007]. Benchmarks show that on this program, incrementalization can reduce asymptotic complexity and can turn $O(n)$ performance into $O(1)$, improving running time by over 4 orders of magnitude on realistic inputs (Chapter 16).

In Chapter 17 we make the following contributions:

- via examples, we motivate extending ILC to remember intermediate results (Sec. 17.2);
- we give a novel proof of correctness for ILC for untyped λ-calculus, based on step-indexed logical relations (Sec. 17.3.3);
- building on top of ILC-style differentiation, we show how to transform untyped higher-order programs to *cache-transfer-style (CTS)* (Sec. 17.3.5);
- we show through formal proofs that programs and derivatives in cache-transfer style *simulate* correctly their non-CTS variants (Sec. 17.3.6);
- we perform performance case studies (in Sec. 17.4) applying (by hand) extension of this technique to Haskell programs, and incrementalize efficiently also programs that do not admit self-maintainable derivatives.

Chapter 18 describes how to extend differentiation to System F. To this end, we extend change structure to allow from changes where source and destination have different types and enable defining more powerful combinators for change structures to be more powerful. While the results in this chapter call for further research, we consider them exciting

Appendix C proposes the first correctness proofs for ILC via operational methods and (step-indexed) logical relations, for simply-typed λ-calculus (without and with general recursion) and for untyped λ-calculus. A later variant of this proof, adapted for use of cache-transfer-style, is presented in Chapter 17, but we believe the presentation in Appendix C might also be interesting for theorists, as it explains how to extend ILC correctness proofs with fewer extraneous complications. Nevertheless, given the similarity between proofs in Appendix C and Chapter 17, we relegate the latter one to an appendix.

Finally, Appendix D shows how to implement all change operations on function changes efficiently, by defunctionalizing functions and function changes rather than using mere closure conversion.

Chapter 11

A tour of differentiation examples

Before formalizing ILC, we show more example of change structures and primitives, to show (a) designs for reusable primitives and their derivatives, (b) to what extent we can incrementalize basic building blocks such as recursive functions and algebraic data types, and (c) to sketch how we can incrementalize collections efficiently. We make no attempt at incrementalizing a complete collection API here; we discuss briefly more complete implementations in Chapter 16 and Sec. 17.4.

To describe these examples informally, we use Haskell notation and **let** polymorphism as appropriate (see Appendix A.2).

We also motivate a few extensions to differentiation that we describe later. As we'll see in this chapter, we'll need to enable some forms of introspection on function changes to manipulate the embedded environments, as we discuss in Appendix D. We will also need ways to remember intermediate results, which we will discuss in Chapter 17. We will also use overly simplified change structures to illustrate a few points.

11.1 Change structures as type-class instances

We encode change structures, as sketched earlier in Sec. 10.3, through a *type class* called *ChangeStruct*. An instance *ChangeStruct t* defines a change type Δt as an associated type and operations $\oplus$, $\ominus$ and $\circledcirc$ are defined as methods. We also define method *oreplace*, such that *oreplace* v_2 produces a *replacement change* from any source to v_2; by default, $v_2 \ominus v_1$ is simply an alias for *oreplace* v_2.

class *ChangeStruct t* **where**
  type Δt
  $(\oplus) :: t \to \Delta t \to t$
  oreplace $:: t \to \Delta t$
  $(\ominus) :: t \to t \to \Delta t$
  $v_2 \ominus v_1 =$ *oreplace* v_2
  $(\circledcirc) :: \Delta t \to \Delta t \to \Delta t$
  $\mathbf{0} :: t \to \Delta t$

In this chapter we will often show change structures where only some methods are defined; in actual implementations we use a type class hierarchy to encode what operations are available, but we collapse this hierarchy here to simplify presentation.

11.2 How to design a language plugin

When adding support for a datatype T, we will strive to define both a change structure and derivatives of introduction and elimination forms for T, since such forms constitute a complete API for using that datatype. However, we will sometimes have to restrict elimination forms to scenarios that can be incrementalized efficiently.

In general, to differentiate a primitive $f : A \to B$ once we have defined a change structure for A, we can start by defining

$$df\ a_1\ da = f\ (a_1 \oplus da) \ominus f\ a_1, \tag{11.1}$$

where da is a valid change from a_1 to a_2. We then try to simplify and rewrite the expression using *equational reasoning*, so that it does not refer to $\ominus$ any more, as far as possible. We can assume that all argument changes are valid, especially if that allows producing faster derivatives; we formalize equational reasoning for valid changes in Sec. 14.1.1. In fact, instead of defining $\ominus$ and simplifying $f\ a_2 \ominus f\ a_1$ to not use it, it is sufficient to produce a change from $f\ a_1$ to $f\ a_2$, even a different one. We write $da_1 =_\Delta da_2$ to mean that changes da_1 and da_2 are equivalent, that is they have the same source and destination. We define this concept properly in Sec. 14.2.

We try to avoid running $\ominus$ on arguments of non-constant size, since it might easily take time linear or superlinear in the argument sizes; if $\ominus$ produces replacement values, it completes in constant time but derivatives invoked on the produced changes are not efficient.

11.3 Incrementalizing a collection API

In this section, we describe a collection API that we incrementalize (partially) in this chapter.

To avoid notation conflicts, we represent lists via datatype *List a*, defined as follows:

data *List a* = *Nil* | *Cons a* (*List a*)

We also consider as primitive operation a standard mapping function *map*. We also support two restricted forms of aggregation: (a) folding over an abelian group via *fold*, similar to how one usually folds over a monoid;[1] (b) list concatenation via *concat*. We will not discuss how to differentiate *concat*, as we reuse existing solutions by Firsov and Jeltsch [2016].

```
singleton :: a → List a
singleton x = Cons x Nil

map :: (a → b) → List a → List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)

fold :: AbelianGroupChangeStruct b ⇒ List b → b
fold Nil = mempty
fold (Cons x xs) = x ⋄ fold xs    -- Where ⋄ is infix for mappend.

concat :: List (List a) → List a
concat = ...
```

While usually *fold* requires only an instance *Monoid b* of type class *Monoid* to aggregate collection elements, our variant of *fold* requires an instance of type class *GroupChangeStruct*, a subclass of *Monoid*. This type class is not used by *fold* itself, but only by its derivative, as we explain in Sec. 11.3.2; nevertheless, we add this stronger constraint to *fold* itself because we forbid derivatives

[1] https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Foldable.html.

with stronger type-class constraints. With this approach, all clients of *fold* can be incrementalized using differentiation.

Using those primitives, one can define further higher-order functions on collections such as *concatMap*, *filter*, *foldMap*. In turn, these functions form the kernel of a collection API, as studied for instance by work on the *monoid comprehension calculus* [Grust and Scholl, 1996a; Fegaras and Maier, 1995; Fegaras, 1999], even if they are not complete by themselves.

$$
\begin{aligned}
&\mathit{concatMap} :: (a \to \mathit{List}\ b) \to \mathit{List}\ a \to \mathit{List}\ b\\
&\mathit{concatMap}\ f = \mathit{concat} \circ \mathit{map}\ f\\
&\mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a\\
&\mathit{filter}\ p = \mathit{concatMap}\ (\lambda x \to \textbf{if}\ p\ x\ \textbf{then}\ \mathit{singleton}\ x\ \textbf{else}\ \mathit{Nil})\\
&\mathit{foldMap} :: \mathit{AbelianGroupChangeStruct}\ b \Rightarrow (a \to b) \to \mathit{List}\ a \to b\\
&\mathit{foldMap}\ f = \mathit{fold} \circ \mathit{map}\ f
\end{aligned}
$$

In first-order DSLs such as SQL, such functionality must typically be added through separate primitives (consider for instance *filter*), while here we can simply *define*, for instance, *filter* on top of *concatMap*, and incrementalize the resulting definitions using differentiation.

Function *filter* uses conditionals, which we haven't discussed yet; we show how to incrementalize *filter* successfully in Sec. 11.6.

11.3.1 Changes to type-class instances?

In this whole chapter, we assume that type-class instances, such as *fold*'s *AbelianGroupChangeStruct* argument, do not undergo changes. Since type-class instances are closed top-level definitions of operations and are canonical for a datatype, it is hard to imagine a change to a type-class instance. On the other hand, type-class instances can be encoded as first-class values. We can for instance imagine a fold taking a unit value and an associative operation as argument. In such scenarios, one needs additional effort to propagate changes to operation arguments, similarly to changes to the function argument to *map*.

11.3.2 Incrementalizing aggregation

Let's now discuss how to incrementalize *fold*. We consider an oversimplified change structure that allows only two sorts of changes: prepending an element to a list or removing the list head of a non-empty list, and study how to incrementalize *fold* for such changes:

$$
\begin{aligned}
&\textbf{data}\ \mathit{ListChange}\ a = \mathit{Prepend}\ a \mid \mathit{Remove}\\
&\textbf{instance}\ \mathit{ChangeStruct}\ (\mathit{List}\ a)\ \textbf{where}\\
&\quad \textbf{type}\ \Delta(\mathit{List}\ a) = \mathit{ListChange}\ a\\
&\quad \mathit{xs} \oplus \mathit{Prepend}\ x = \mathit{Cons}\ x\ \mathit{xs}\\
&\quad (\mathit{Cons}\ x\ \mathit{xs}) \oplus \mathit{Remove} = \mathit{xs}\\
&\quad \mathit{Nil} \oplus \mathit{Remove} = \mathit{error}\ \texttt{"Invalid change"}\\
&\mathit{dfold}\ \mathit{xs}\ (\mathit{Prepend}\ x) = \ldots
\end{aligned}
$$

Removing an element from an empty list is an invalid change, hence it is safe to give an error in that scenario as mentioned when introducing $\oplus$ (Sec. 10.3).

By using equational reasoning as suggested in Sec. 11.2, starting from Eq. (11.1), one can show formally that $\mathit{dfold}\ \mathit{xs}\ (\mathit{Prepend}\ x)$ should be a change that, in a sense, "adds" x to the result using group operations:

$$
\begin{aligned}
& \mathit{dfold}\ \mathit{xs}\ (\mathit{Prepend}\ x) \\
=_\Delta\ & \mathit{fold}\ (\mathit{xs} \oplus \mathit{Prepend}\ x) \ominus \mathit{fold}\ \mathit{xs} \\
=\ & \mathit{fold}\ (\mathit{Cons}\ x\ \mathit{xs}) \ominus \mathit{fold}\ \mathit{xs} \\
=\ & (x \diamond \mathit{fold}\ \mathit{xs}) \ominus \mathit{fold}\ \mathit{xs}
\end{aligned}
$$

Similarly, $\mathit{dfold}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{Remove}$ should instead "subtract" x from the result:

$$
\begin{aligned}
& \mathit{dfold}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{Remove} \\
=_\Delta\ & \mathit{fold}\ (\mathit{Cons}\ x\ \mathit{xs} \oplus \mathit{Remove}) \ominus \mathit{fold}\ (\mathit{Cons}\ x\ \mathit{xs}) \\
=\ & \mathit{fold}\ \mathit{xs} \ominus \mathit{fold}\ (\mathit{Cons}\ x\ \mathit{xs}) \\
=\ & \mathit{fold}\ \mathit{xs} \ominus (x \diamond \mathit{fold}\ \mathit{xs})
\end{aligned}
$$

As discussed, using $\ominus$ is fast enough on, say, integers or other primitive types, but not in general. To avoid using $\ominus$ we must rewrite its invocation to an equivalent expression. In this scenario we can use group changes for abelian groups, and restrict *fold* to situations where such changes are available.

$$
\begin{aligned}
& \mathit{dfold} :: \mathit{AbelianGroupChangeStruct}\ b \Rightarrow \mathit{List}\ b \to \Delta(\mathit{List}\ b) \to \Delta b \\
& \mathit{dfold}\ \mathit{xs}\ (\mathit{Prepend}\ x) = \mathit{inject}\ x \\
& \mathit{dfold}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{Remove} = \mathit{inject}\ (\mathit{invert}\ x) \\
& \mathit{dfold}\ \mathit{Nil}\ \mathit{Remove} = \mathit{error}\ \texttt{"Invalid change"}
\end{aligned}
$$

To support group changes we define the following type classes to model abelian groups and group change structures, omitting APIs for more general groups. *AbelianGroupChangeStruct* only requires that group elements of type g can be converted into changes (type Δg), allowing change type Δg to contain other sorts of changes.

$$
\begin{aligned}
& \textbf{class}\ \mathit{Monoid}\ g \Rightarrow \mathit{AbelianGroup}\ g\ \textbf{where} \\
& \quad \mathit{invert} :: g \to g \\
& \textbf{class}\ (\mathit{AbelianGroup}\ a, \mathit{ChangeStruct}\ a) \Rightarrow \\
& \qquad \mathit{AbelianGroupChangeStruct}\ a\ \textbf{where} \\
& \quad \text{-- Inject group elements into changes. Law:} \\
& \quad \text{-- } a \oplus \mathit{inject}\ b = a \diamond b \\
& \quad \mathit{inject} :: a \to \Delta a
\end{aligned}
$$

Chapter 16 discusses how we can use group changes without assuming a single group is defined on elements, but here we simply select the canonical group as chosen by type-class resolution. To use a different group, as usual, one defines a different but isomorphic type via the Haskell **newtype** construct. As a downside, derivatives **newtype** constructors must convert changes across different representations.

Rewriting $\ominus$ away can also be possible for other specialized folds, though sometimes they can be incrementalized directly; for instance *map* can be written as a fold. Incrementalizing *map* for the insertion of x into xs requires simplifying $\mathit{map}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) \ominus \mathit{map}\ f\ \mathit{xs}$. To avoid $\ominus$ we can rewrite this change statically to $\mathit{Insert}\ (f\ x)$; indeed, we can incrementalize *map* also for more realistic change structures.

Associative tree folds Other usages of fold over sequences produce result type of small bounded size (such as integers). In this scenario, one can incrementalize the given fold efficiently using $\ominus$ instead of relying on group operations. For such scenarios, one can design a primitive

$$\mathit{foldMonoid} :: \mathit{Monoid}\ a \Rightarrow \mathit{List}\ a \Rightarrow a$$

for *associative tree folds*, that is, a function that folds over the input sequence using a *monoid* (that is, an associative operation with a unit). For efficient incrementalization, *foldMonoid*'s intermediate results should form a *balanced* tree and updating this tree should take *logarithmic* time: one approach to ensure this is to represent the input sequence itself using a balanced tree, such as a finger tree [Hinze and Paterson, 2006].

Various algorithms store intermediate results of folding inside an input balanced tree, as described by Cormen et al. [2001, Ch. 14] or by Hinze and Paterson [2006]. But intermediate results can also be stored outside the input tree, as is commonly done using self-adjusting computation [Acar, 2005, Sec. 9.1], or as can be done in our setting. While we do not use such folds, we describe the existing algorithms briefly and sketch how to integrate them in our setting.

Function *foldMonoid* must record the intermediate results, and the derivative *dfoldMonoid* must propagate input changes to affected intermediate results.

[2] To study time complexity of input change propagation, it is useful to consider the *dependency graph* of intermediate results: in this graph, an intermediate result v_1 has an arc to intermediate result v_2 if and only if computing v_1 depends on v_2. To ensure *dfoldMonoid* is efficient, the dependency graph of intermediate results from *foldMonoid* must form a balanced tree of logarithmic height, so that changes to a leaf only affect a logarithmic number of intermediate results.

In contrast, implementing *foldMonoid* using *foldr* on a list produces an unbalanced graph of intermediate results. For instance, take input list $xs = [1\,..\,8]$, containing numbers from 1 to 8, and assume a given monoid. Summing them with *foldr* $(\diamond)$ *mempty xs* means evaluating

$$1 \diamond (2 \diamond (3 \diamond (4 \diamond (5 \diamond (6 \diamond (7 \diamond (8 \diamond \mathit{mempty}))))))).$$

Then, a change to the last element of input *xs* affects all intermediate results, hence incrementalization takes at least $O(n)$. In contrast, using *foldAssoc* on *xs* should evaluate a balanced tree similar to

$$((1 \diamond 2) \diamond (3 \diamond 4)) \diamond ((5 \diamond 6) \diamond (7 \diamond 8)),$$

so that any individual change to a leave, insertion or deletion only affects $O(\log n)$ intermediate results (where n is the sequence size). Upon modifications to the tree, one must ensure that the balancing is stable [Acar, 2005, Sec. 9.1]. In other words, altering the tree (by inserting or removing an element) must only alter $O(\log n)$ nodes.

We have implemented associative tree folds on very simple but unbalanced tree structures; we believe they could be implemented and incrementalized over balanced trees representing sequences, such as finger trees or random access zippers [Headley and Hammer, 2016], but doing so requires transforming their implementation of their data structure to cache-transfer style (CTS) (Chapter 17). We leave this for future work, together with an automated implementation of CTS transformation.

11.3.3 Modifying list elements

In this section, we consider another change structure on lists that allows expressing changes to individual elements. Then, we present *dmap*, derivative of *map* for this change structure. Finally, we sketch informally the correctness of *dmap*, which we prove formally in Example 14.1.1.

We can then represent changes to a list $(\mathit{List}\ a)$ as a list of changes $(\mathit{List}\ (\Delta a))$, one for each element. A list change *dxs* is valid for source *xs* if they have the same length and each element change is valid for its corresponding element. For this change structure we can define $\oplus$ and $\circledcirc$, but not a total $\ominus$: such list changes can't express the difference between two lists of different lengths. Nevertheless, this change structure is sufficient to define derivatives that act correctly

[2] We discuss in Chapter 17 how base functions communicate results to derivatives.

on the changes that can be expressed. We can describe this change structure in Haskell using a type-class instance for class *ChangeStruct*:

```
instance ChangeStruct (List a) where
  type Δ(List a) = List (Δa)
  Nil ⊕ Nil = Nil
  (Cons x xs) ⊕ (Cons dx dxs) = Cons (x ⊕ xs) (dx ⊕ dxs)
  _ ⊕ _ = Nil
```

The following *dmap* function is a derivative for the standard *map* function (included for reference) and the given change structure. We discuss derivatives for recursive functions in Sec. 15.1.

```
map : (a → b) → List a → List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
dmap : (a → b) → Δ(a → b) → List a → ΔList a → ΔList b
  -- A valid list change has the same length as the base list:
dmap f df Nil Nil = Nil
dmap f df (Cons x xs) (Cons dx dxs) =
  Cons (df x dx) (dmap f df xs dxs)
  -- Remaining cases deal with invalid changes, and a dummy
  -- result is sufficient.
dmap f df xs dxs = Nil
```

Function *dmap* is a correct derivative of *map* for this change structure, according to Slogan 10.3.3: we sketch an informal argument by induction. The equation for $dmap\ f\ df\ Nil\ Nil$ returns *Nil*, a valid change from initial to updated outputs, as required. In the equation for $dmap\ f\ df\ (Cons\ x\ xs)\ (Cons\ dx\ dxs)$ we compute changes to the head and tail of the result, to produce a change from $map\ f\ (Cons\ x\ xs)$ to $map\ (f \oplus df)\ (Cons\ x\ xs \oplus Cons\ dx\ dxs)$. To this end, (a) we use $df\ x\ dx$ to compute a change to the head of the result, from $f\ x$ to $(f \oplus df)\ (x \oplus dx)$; (b) we use $dmap\ f\ df\ xs\ dxs$ recursively to compute a change to the tail of the result, from $map\ f\ xs$ to $map\ (f \oplus df)\ (xs \oplus dxs)$; (c) we assemble changes to head and tail with *Cons* into a change from In other words, *dmap* turns input changes to output changes correctly according to our Slogan 10.3.3: it is a correct derivative for *map* according to this change structure. We have reasoned informally; we formalize this style of reasoning in Sec. 14.1. Crucially, our conclusions only hold if input changes are valid, hence term $map\ f\ xs \oplus dmap\ f\ df\ xs\ dxs$ is not denotationally equal to $map\ (f \oplus df)\ (xs \oplus dxs)$ for arbitrary change environments: these two terms only evaluate to the same result for valid input changes.

Since this definition of *dmap* is a correct derivative, we could use it in an incremental DSL for list manipulation, together with other primitives. Because of limitations we describe next, we will use instead improved language plugins for sequences.

11.3.4 Limitations

We have shown simplified list changes, but they have a few limitations. Fixing those requires more sophisticated definitions.

As discussed, our list changes intentionally forbid changing the length of a list. And our definition of *dmap* has further limitations: a change to a list of n elements takes size $O(n)$, even when most elements do not change, and calling $dmap\ f\ df$ on it requires n calls to df. This is only faster if df is faster than f, but adds no further speedup.

We can describe instead a change to an arbitrary list element x in xs by giving the change dx and the position of x in xs. A list change is then a sequence of such changes:

type $\Delta(List\ a) = List\ (AtomicChange\ a)$
data $AtomicChange\ a = Modify\ \mathbb{Z}\ (\Delta a)$

However, fetching the i-th list element still takes time linear in i: we need a better representation of sequences. In next section, we switch to a change structure on sequences defined by finger trees [Hinze and Paterson, 2006], following Firsov and Jeltsch [2016].

11.4 Efficient sequence changes

Firsov and Jeltsch [2016] define an efficient representation of list changes in a framework similar to ILC, and incrementalize selected operations over this change structure. They also provide combinators to assemble further operations on top of the provided ones. We extend their framework to handle function changes and generate derivatives for all functions that can be expressed in terms of the primitives.

Conceptually, a change for type *Sequence a* is a sequence of atomic changes. Each atomic change inserts one element at a given position, or removes one element, or changes an element at one position.[3]

data *SeqSingleChange a*
  $= Insert \quad \{idx :: \mathbb{Z}, x :: a\}$
  $\mid Remove \quad \{idx :: \mathbb{Z}\}$
  $\mid ChangeAt\ \{idx :: \mathbb{Z}, dx :: \Delta a\}$
data $SeqChange\ a = Sequence\ (SeqSingleChange\ a)$
type $\Delta(Sequence\ a) = SeqChange\ a$

We use Firsov and Jeltsch's variant of this change structure in Sec. 17.4.

11.5 Products

It is also possible to define change structures for arbitrary sum and product types, and to provide derivatives for introduction and elimination forms for such datatypes. In this section we discuss products, in the next section sums.

We define a simple change structure for product type $A \times B$ from change structures for A and B, similar to change structures for environments: operations act pointwise on the two components.

instance $(ChangeStruct\ a, ChangeStruct\ b) \Rightarrow ChangeStruct\ (a, b)$ **where**
  type $\Delta(a, b) = (\Delta a, \Delta b)$
  $(a, b) \oplus (da, db) = (a \oplus da, b \oplus db)$
  $(a_2, b_2) \ominus (a_1, b_1) = (a_2 \ominus a_1, b_2 \ominus b_1)$
  $oreplace\ (a_2, b_2) = (oreplace\ a_2, oreplace\ b_2)$
  $\mathbf{0}_{a,b} = (\mathbf{0}_a, \mathbf{0}_b)$
  $(da_1, db_1) \circledcirc (da_2, db_2) = (da_1 \circledcirc da_2, db_1 \circledcirc db_2)$

Through equational reasoning as in Sec. 11.2, we can also compute derivatives for basic primitives on product types, both the introduction form (that we alias as *pair*) and the elimination forms *fst* and *snd*. We just present the resulting definitions:

$pair\ a\ b = (a, b)$
$dpair\ a\ da\ b\ db = (da, db)$

[3] Firsov and Jeltsch [2016] and our actual implementation allow changes to multiple elements.

```
fst (a, b) = a
snd (a, b) = b
dfst :: Δ(a, b) → Δa
dfst (da, db) = da
dsnd :: Δ(a, b) → Δb
dsnd (da, db) = db

uncurry :: (a → b → c) → (a, b) → c
uncurry f (a, b) = f a b
duncurry :: (a → b → c) → Δ(a → b → c) → (a, b) → Δ(a, b) → Δc
duncurry f df (x, y) (dx, dy) = df x dx y dy
```

One can also define n-ary products in a similar way. However, a product change contains as many entries as a product.

11.6 Sums, pattern matching and conditionals

In this section we define change structures for sum types, together with the derivative of their introduction and elimination forms. We also obtain support for booleans (which can be encoded as sum type $1 + 1$) and conditionals (which can be encoded in terms of elimination for sums). We have mechanically proved correctness of this change structure and derivatives, but we do not present the tedious details in this thesis and refer to our Agda formalization.

Changes structures for sums are more challenging than ones for products. We can define them, but in many cases we can do better with specialized structures. Nevertheless, such changes are useful in some scenarios.

In Haskell, sum types $a + b$ are conventionally defined via datatype *Either a b*, with introduction forms *Left* and *Right* and elimination form *either* which will be our primitives:

```
data Either a b = Left a | Right b

either :: (a → c) → (b → c) → Either a b → c
either f g (Left a) = f a
either f g (Right b) = g b
```

We can define the following change structure.

```
data EitherChange a b
  = LeftC (Δa)
  | RightC (Δb)
  | EitherReplace (Either a b)

instance (ChangeStruct a, ChangeStruct b) ⇒
         ChangeStruct (Either a b) where
  type Δ(Either a b) = EitherChange a b
  Left a  ⊕ LeftC da        = Left (a ⊕ da)
  Right b ⊕ RightC db       = Right (b ⊕ db)
  Left _  ⊕ RightC _        = error "Invalid change!"
  Right _ ⊕ LeftC _         = error "Invalid change!"
  _       ⊕ EitherReplace e₂ = e₂
  oreplace = EitherReplace

  0_{Left a} = LeftC 0_a
  0_{Right a} = RightC 0_a
```

Changes to a sum value can either keep the same "branch" (left or right) and modify the contained value, or replace the sum value with another one altogether. Specifically, change $LeftC\ da$ is valid from $Left\ a_1$ to $Left\ a_2$ if da is valid from a_1 to a_2. Similarly, change $RightC\ db$ is valid from $Right\ b_1$ to $Right\ b_2$ if db is valid from b_1 to b_2. Finally, replacement change $EitherReplace\ e_2$ is valid from e_1 to e_2 for any e_1.

Using Eq. (11.1), we can then obtain definitions for derivatives of primitives *Left*, *Right* and *either*. The resulting code is as follows:

$$
\begin{aligned}
&dLeft :: a \to \Delta a \to \Delta(Either\ a\ b)\\
&dLeft\ a\ da = LeftC\ da\\
&dRight :: b \to \Delta b \to \Delta(Either\ a\ b)\\
&dRight\ b\ db = RightC\ db\\
&deither ::\\
&\quad (NilChangeStruct\ a, NilChangeStruct\ b, DiffChangeStruct\ c) \Rightarrow\\
&\quad (a \to c) \to \Delta(a \to c) \to (b \to c) \to \Delta(b \to c) \to\\
&\quad Either\ a\ b \to \Delta(Either\ a\ b) \to \Delta c\\
&deither\ f\ df\ g\ dg\ (Left\ a)\ (LeftC\ da) = df\ a\ da\\
&deither\ f\ df\ g\ dg\ (Right\ b)\ (RightC\ db) = dg\ b\ db\\
&deither\ f\ df\ g\ dg\ e_1\ (EitherReplace\ e_2) =\\
&\quad either\ (f \oplus df)\ (g \oplus dg)\ e_2 \ominus either\ f\ g\ e_1\\
&deither\ \_\ \_\ \_\ \_\ \_\ \_ = error\ \texttt{"Invalid sum change"}
\end{aligned}
$$

We show only one case of the derivation of *deither* as an example:

$$
\begin{aligned}
&\quad deither\ f\ df\ g\ dg\ (Left\ a)\ (LeftC\ da)\\
&=_{\Delta} \quad \{\text{ using variants of Eq. (11.1) for multiple arguments }\}\\
&\quad either\ (f \oplus df)\ (g \oplus dg)\ (Left\ a \oplus LeftC\ da) \ominus either\ f\ g\ (Left\ a)\\
&= \quad \{\text{ simplify } \oplus \}\\
&\quad either\ (f \oplus df)\ (g \oplus dg)\ (Left\ (a \oplus da)) \ominus either\ f\ g\ (Left\ a)\\
&= \quad \{\text{ simplify } either \}\\
&\quad (f \oplus df)\ (a \oplus da) \ominus f\ a\\
&=_{\Delta} \quad \{\text{ because } df \text{ is a valid change for } f, \text{ and } da \text{ for } a \}\\
&\quad df\ a\ da
\end{aligned}
$$

Unfortunately, with this change structure a change from $Left\ a_1$ to $Right\ b_2$ is simply a replacement change, so derivatives processing it must recompute results from scratch. In general, we cannot do better, since there need be no shared data between two branches of a datatype. We need to find specialized scenarios where better implementations are possible.

Extensions, changes for ADTs and future work In some cases, we consider changes for type $Either\ a\ b$, where a and b both contain some other type c. Take again lists: a change from list as to list $Cons\ a_2\ as$ should simply say that we prepend a_2 to the list. In a sense, we are just using a change structure from type $List\ a$ to $(a, List\ a)$. More in general, if change das from as_1 to as_2 is small, a change from list as_1 to list $Cons\ a_2\ as_2$ should simply "say" that we prepend a_2 and that we modify as_1 into as_2, and similarly for removals.

In Sec. 18.4, we suggest how to construct such change structures, based on the concept of *polymorphic change structure*, where changes have source and destination of different types. Based on initial experiments, we believe one could develop these constructions into a powerful combinator language for change structures. In particular, it should be possible to build change structures for lists

similar to the ones in Sec. 11.3.2. Generalizing beyond lists, similar systematic constructions should be able to represent insertions and removals of one-hole contexts [McBride, 2001] for arbitrary algebraic datatypes (ADTs); for ADTs representing balanced data structures, such changes could enable efficient incrementalization in many scenarios. However, many questions remain open, so we leave this effort for future work.

11.6.1 Optimizing *filter*

In Sec. 11.3 we have defined *filter* using a conditional, and now we have just explained that in general conditionals are inefficient! This seems pretty unfortunate. But luckily, we can optimize *filter* using equational reasoning.

Consider again the earlier definition of *filter*:

$$\begin{aligned}&\mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a\\&\mathit{filter}\ p = \mathit{concatMap}\ (\lambda x \to \textbf{if}\ p\ x\ \textbf{then}\ \mathit{singleton}\ x\ \textbf{else}\ \mathit{Nil})\end{aligned}$$

As explained, we can encode conditionals using *either* and differentiate the resulting program. However, if $p\ x$ changes from *True* to *False*, or viceversa (that is, for all elements x for which $dp\ x\ dx$ is a non-nil change), we must compute $\ominus$ at runtime. However, $\ominus$ will compare empty list *Nil* with a singleton list produced by $\mathit{singleton}\ x$ (in one direction or the other). We can have $\ominus$ detect this situation at runtime. But since the implementation of *filter* is known statically, we can optimize this code at runtime, rewriting $\mathit{singleton}\ x \ominus \mathit{Nil}$ to $\mathit{Insert}\ x$, and $\mathit{Nil} \ominus \mathit{singleton}\ x$ to *Remove*. To enable this optimization in *dfilter*, we need to inline the function that *filter* passes as argument to *concatMap* and all the functions it calls except p. Moreover, we need to case-split on possible return values for $p\ x$ and $dp\ x\ dx$. We omit the steps because they are both tedious and standard.

It appears in principle possible to automate such transformations by adding domain-specific knowledge to a sufficiently smart compiler, though we have made no attempt at an actual implementation. It would be first necessary to investigate further classes of examples where optimizations are applicable. Sufficiently smart compilers are rare, but since our approach produces purely functional programs we have access to GHC and HERMIT [Farmer et al., 2012]. An interesting alternative (which does have some support for side effects) is LMS [Rompf and Odersky, 2010] and Delite [Brown et al., 2011]. We leave further investigation for future work.

11.7 Chapter conclusion

In this chapter we have toured what can and cannot be incrementalized using differentiation, and how using higher-order functions allows defining generic primitives to incrementalize.

Chapter 12

Changes and differentiation, formally

To support incrementalization, in this chapter we introduce differentiation and formally prove it correct. That is, we prove that $[\![\mathcal{D}[\![t]\!]]\!]$ produces derivatives. As we explain in Sec. 12.1.2, derivatives transform valid input changes into valid output changes (Slogan 10.3.3). Hence, we define what are valid changes (Sec. 12.1). As we'll explain in Sec. 12.1.4, validity is a logical relation. As we explain in Sec. 12.2.2, our correctness theorem is the *fundamental property* for the validity logical relation, proved by induction over the structure of terms. Crucial definitions or derived facts are summarized in Fig. 12.1. Later, in Chapter 13 we study consequences of correctness and change operations.

All definitions and proofs in this and next chapter is mechanized in Agda, except where otherwise indicated. To this writer, given these definitions all proofs have become straightforward and unsurprising. Nevertheless, first obtaining these proofs took a while. So we typically include full proofs. We also believe these proofs clarify the meaning and consequences of our definitions. To make proofs as accessible as possible, we try to provide enough detail that our target readers can follow along *without* pencil and paper, at the expense of making our proofs look longer than they would usually be. As we target readers proficient with STLC (but not necessarily proficient with logical relations), we'll still omit routine steps needed to reason on STLC, such as typing derivations or binding issues.

12.1 Changes and validity

In this section we introduce formally (a) a description of changes; (b) a definition of which changes are valid. We have already introduced informally in Chapter 10 these notions and how they fit together. We next define the same notions formally, and deduce their key properties. Language plugins extend these definitions for base types and constants that they provide.

To formalize the notion of changes for elements of a set V, we define the notion of *basic change structure* on V.

Definition 12.1.1 (Basic change structures)
A basic change structure on set V, written $\widetilde{V}$, comprises:

(a) a change set ΔV

$\boxed{\Delta\tau}$

$$\Delta\iota = \dots$$
$$\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau$$

(a) Change types (Definition 12.1.17).

$\boxed{\Delta\Gamma}$

$$\Delta\varepsilon = \varepsilon$$
$$\Delta\,(\Gamma, x : \tau) = \Delta\Gamma, x : \tau, dx : \Delta\tau$$

(b) Change contexts (Definition 12.1.23).

$\boxed{\mathcal{D}[\![\, t \,]\!]}$

$$\begin{aligned}
\mathcal{D}[\![\, \lambda(x : \sigma) \to t \,]\!] &= \lambda(x : \sigma)\ (dx : \Delta\sigma) \to \mathcal{D}[\![\, t \,]\!] \\
\mathcal{D}[\![\, s\ t \,]\!] &= \mathcal{D}[\![\, s \,]\!]\ \ t\ \mathcal{D}[\![\, t \,]\!] \\
\mathcal{D}[\![\, x \,]\!] &= dx \\
\mathcal{D}[\![\, c \,]\!] &= \mathcal{D}^C[\![\, c \,]\!]
\end{aligned}$$

(c) Differentiation (Definition 10.5.1).

$$\frac{\Gamma \vdash t : \tau}{\Delta\Gamma \vdash \mathcal{D}[\![\, t \,]\!] : \Delta\tau}\ \textsc{Derive}$$

(d) Differentiation typing (Lemma 12.2.8).

$\boxed{dv \triangleright v_1 \hookrightarrow v_2 : \tau \text{ with } v_1, v_2 : [\![\, \tau \,]\!], dv : [\![\, \Delta\tau \,]\!]}$

$$dv \triangleright v_1 \hookrightarrow v_2 : \iota = \dots$$
$$df \triangleright f_1 \hookrightarrow f_2 : \sigma \to \tau = \forall da \triangleright a_1 \hookrightarrow a_2 : \sigma.\ \ df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : \tau$$

$\boxed{d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma \text{ with } \rho_1, \rho_2 : [\![\, \Gamma \,]\!], d\rho : [\![\, \Delta\Gamma \,]\!]}$

$$\frac{}{\varnothing \triangleright \varnothing \hookrightarrow \varnothing : \varepsilon} \qquad \frac{d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma \qquad da \triangleright a_1 \hookrightarrow a_2 : \tau}{(d\rho, x = a_1, dx = da) \triangleright (\rho_1, x = a_1) \hookrightarrow (\rho_2, x = a_2) : \Gamma, x : \tau}$$

(e) Validity (Definitions 12.1.21 and 12.1.24).

If $\Gamma \vdash t : \tau$ then

$$\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma.\ \ [\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ \ d\rho \triangleright [\![\, t \,]\!]\ \ \rho_1 \hookrightarrow [\![\, t \,]\!]\ \ \rho_2 : \tau.$$

(f) Correctness of $\mathcal{D}[\![\, - \,]\!]$ (from Theorem 12.2.2).

Figure 12.1: Defining differentiation and proving it correct. The rest of this chapter explains and motivates the above definitions.

(b) a ternary *validity* relation $dv \triangleright v_1 \hookrightarrow v_2 : V$, for $v_1, v_2 \in V$ and $dv \in \Delta V$, that we read as "dv is a valid change from source v_1 to destination v_2 (on set V)". □

Example 12.1.2
In Examples 10.3.1 and 10.3.2 we exemplified informally change types and validity on naturals, integers and bags. We define formally basic change structures on naturals and integers. Compared to validity for integers, validity for naturals ensures that the destination $v_1 + dv$ is again a natural. For instance, given source $v_1 = 1$, change $dv = -2$ is valid (with destination $v_2 = -1$) only on integers, not on naturals. □

Definition 12.1.3 (Basic change structure on integers)
Basic change structure $\widetilde{\mathbb{Z}}$ on integers has integers as changes ($\Delta\mathbb{Z} = \mathbb{Z}$) and the following validity judgment.

$$\frac{}{dv \triangleright v_1 \hookrightarrow v_1 + dv : \mathbb{Z}}$$

□

Definition 12.1.4 (Basic change structure on naturals)
Basic change structure $\widetilde{\mathbb{N}}$ on naturals has integers as changes ($\Delta\mathbb{N} = \mathbb{Z}$) and the following validity judgment.

$$\frac{v_1 + dv \geqslant 0}{dv \triangleright v_1 \hookrightarrow v_1 + dv : \mathbb{N}}$$

□

Intuitively, we can think of a valid change from v_1 to v_2 as a graph *edge* from v_1 to v_2, so we'll often use graph terminology when discussing changes. This intuition is robust and can be made fully precise.[1] More specifically, a basic change structure on V can be seen as a directed multigraph, having as vertexes the elements of V, and as edges from v_1 to v_2 the valid changes dv from v_1 to v_2. This is a multigraph because our definition allows multiple edges between v_1 and v_2.

A change dv can be valid from v_1 to v_2 and from v_3 to v_4, but we'll still want to talk about *the* source and *the* destination of a change. When we talk about a change dv valid from v_1 to v_2, value v_1 is dv's source and v_2 is dv's destination. Hence we'll systematically quantify theorems over valid changes dv with their sources v_1 and destination v_2, using the following notation.[2]

Notation 12.1.5 (Quantification over valid changes)
We write

$$\forall dv \triangleright v_1 \hookrightarrow v_2 : V.\, P,$$

and say "for all (valid) changes dv from v_1 to v_2 on set V we have P", as a shortcut for

$$\forall v_1, v_2 \in V, dv \in \Delta V, \text{ if } dv \triangleright v_1 \hookrightarrow v_2 : V \text{ then } P.$$

Since we focus on valid changes, we'll omit the word "valid" when clear from context. In particular, a change from v_1 to v_2 is necessarily valid. □

[1] See for instance Robert Atkey's blog post [Atkey, 2015] or Yufei Cai's PhD thesis [Cai, 2017].

[2] If you prefer, you can tag a change with its source and destination by using a triple, and regard the whole triple (v_1, dv, v_2) as a change. Mathematically, this gives the correct results, but we'll typically not use such triples as changes in programs for performance reasons.

We can have multiple basic change structures on the same set.

Example 12.1.6 (Replacement changes)
For instance, for any set V we can talk about *replacement changes* on V: a replacement change $dv = !v_2$ for a value $v_1 : V$ simply specifies directly a new value $v_2 : V$, so that $!v_2 \triangleright v_1 \hookrightarrow v_2 : V$. We read ! as the "bang" operator.

A basic change structure can decide to use only replacement changes (which can be appropriate for primitive types with values of constant size), or to make ΔV a sum type allowing both replacement changes and other ways to describe a change (as long as we're using a language plugin that adds sum types). □

Nil changes Just like integers have a null element 0, among changes there can be nil changes:

Definition 12.1.7 (Nil changes)
We say that $dv : \Delta V$ is a nil change for $v : V$ if $dv \triangleright v \hookrightarrow v : V$. □

For instance, 0 is a nil change for any integer number n. However, in general a change might be nil for an element but not for another. For instance, the replacement change !6 is a nil change on 6 but not on 5.

We'll say more on nil changes in Sec. 12.1.2 and 13.2.1.

12.1.1 Function spaces

Next, we define a basic change structure that we call $\widetilde{A} \to \widetilde{B}$ for an arbitrary function space $A \to B$, assuming we have basic change structures for both A and B. We claimed earlier that valid function changes map valid input changes to valid output changes. We make this claim formal through next definition.

Definition 12.1.8 (Basic change structure on $A \to B$)
Given basic change structures on A and B, we define a basic change structure on $A \to B$ that we write $\widetilde{A} \to \widetilde{B}$ as follows:

(a) Change set $\Delta(A \to B)$ is $A \to \Delta A \to \Delta B$.

(b) Function change df is valid from f_1 to f_2 (that is, $df \triangleright f_1 \hookrightarrow f_2 : A \to B$) if and only if, for all valid input changes $da \triangleright a_1 \hookrightarrow a_2 : A$, value $df\ a_1\ da$ is a valid output change from $f_1\ a_1$ to $f_2\ a_2$ (that is, $df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B$). □

Notation 12.1.9 (Applying function changes to changes)
When reading out $df\ a_1\ da$ we'll often talk for brevity about applying df to da, leaving da's source a_1 implicit when it can be deduced from context. □

We'll also consider valid changes df for curried n-ary functions. We show what their validity means for curried binary functions $f : A \to B \to C$. We omit similar statements for higher arities, as they add no new ideas.

Lemma 12.1.10 (Validity on $A \to B \to C$)
For any basic change structures $\widetilde{A}$, $\widetilde{B}$ and $\widetilde{C}$, function change $df : \Delta(A \to B \to C)$ is valid from f_1 to f_2 (that is, $df \triangleright f_1 \hookrightarrow f_2 : A \to B \to C$) *if and only if* applying df to valid input changes $da \triangleright a_1 \hookrightarrow a_2 : A$ and $db \triangleright b_1 \hookrightarrow b_2 : B$ gives a valid output change

$$df\ a_1\ da\ b_1\ db \triangleright f\ a_1\ b_1 \hookrightarrow f\ a_2\ b_2 : C.$$

□

Proof. The equivalence follows from applying the definition of function validity of df twice.

That is: function change df is valid ($df \triangleright f_1 \hookrightarrow f_2 : A \to (B \to C)$) if and only if it maps valid input change $da \triangleright a_1 \hookrightarrow a_2 : A$ to valid output change

$$df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B \to C.$$

In turn, $df\ a_1\ da$ is a function change, which is valid if and only if it maps valid input change $db \triangleright b_1 \hookrightarrow b_2 : B$ to

$$df\ a_1\ da\ b_1\ db \triangleright f\ a_1\ b_1 \hookrightarrow f\ a_2\ b_2 : C$$

as required by the lemma. □

12.1.2 Derivatives

Among valid function changes, derivatives play a central role, especially in the statement of correctness of differentiation.

Definition 12.1.11 (Derivatives)
Given function $f : A \to B$, function $df : A \to \Delta A \to \Delta B$ is a derivative for f if, for all changes da from a_1 to a_2 on set A, change $df\ a_1\ da$ is valid from $f\ a_1$ to $f\ a_2$. □

However, it follows that derivatives are nil function changes:

Lemma 12.1.12 (Derivatives as nil function changes)
Given function $f : A \to B$, function $df : \Delta(A \to B)$ is a derivative of f if and only if df is a nil change of f ($df \triangleright f \hookrightarrow f : A \to B$). □

Proof. First we show that nil changes are derivatives. First, a nil change $df \triangleright f \hookrightarrow f : A \to B$ has the right type to be a derivative, because $A \to \Delta A \to \Delta B = \Delta(A \to B)$. Since df is a nil change from f to f, by definition it maps valid input changes $da \triangleright a_1 \hookrightarrow a_2 : A$ to valid output changes $df\ a_1\ da \triangleright f\ a_1 \hookrightarrow f\ a_2 : B$. Hence df is a derivative as required.

In fact, all proof steps are equivalences, and by tracing them backward, we can show that derivatives are nil changes: Since a derivative df maps valid input changes $da \triangleright a_1 \hookrightarrow a_2 : A$ to valid output changes $df\ a_1\ da \triangleright f\ a_1 \hookrightarrow f\ a_2 : B$, df is a change from f to f as required. □

Applying derivatives to nil changes gives again nil changes. This fact is useful when reasoning on derivatives. Its proof is a useful exercise on validity.

Lemma 12.1.13 (Derivatives preserve nil changes)
For any basic change structures $\widetilde{A}$ and $\widetilde{B}$, if function change $df : \Delta(A \to B)$ is a derivative of $f : A \to B$ ($df \triangleright f \hookrightarrow f : A \to B$) then applying df to an arbitrary input nil change $da \triangleright a \hookrightarrow a : A$ gives a nil change

$$df\ a\ da \triangleright f\ a \hookrightarrow f\ a : B.$$

□

Proof. Just rewrite the definition of derivatives (Lemma 12.1.12) using the definition of validity of df.

In detail, by definition of validity for function changes (Definition 12.1.8), $df \triangleright f_1 \hookrightarrow f_2 : A \to B$ means that from $da \triangleright a_1 \hookrightarrow a_2 : A$ follows $df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B$. Just substitute $f_1 = f_2 = f$ and $a_1 = a_2 = a$ to get the required implication. □

Also derivatives of curried n-ary functions f preserve nil changes. We only state this formally for curried binary functions $f : A \to B \to C$; higher arities require no new ideas.

Lemma 12.1.14 (Derivatives preserve nil changes on $A \to B \to C$)
For any basic change structures $\widetilde{A}$, $\widetilde{B}$ and $\widetilde{C}$, if function change $df : \Delta(A \to B \to C)$ is a derivative of $f : A \to B \to C$ then applying df to nil changes $da \triangleright a \hookrightarrow a : A$ and $db \triangleright b \hookrightarrow b : B$ gives a nil change

$$df\ a\ da\ b\ db \triangleright f\ a\ b \hookrightarrow f\ a\ b : C.$$

□

Proof. Similarly to validity on $A \to B \to C$ (Lemma 12.1.10), the thesis follows by applying twice the fact that derivatives preserve nil changes (Lemma 12.1.13).

In detail, since derivatives preserve nil changes, df is a derivative only if for all $da \triangleright a \hookrightarrow a : A$ we have $df\ a\ da \triangleright f\ a \hookrightarrow f\ a : B \to C$. But then, $df\ a\ da$ is a nil change, that is a derivative, and since it preserves nil changes, df is a derivative only if for all $da \triangleright a \hookrightarrow a : A$ and $db \triangleright b \hookrightarrow b : B$ we have $df\ a\ da\ b\ db \triangleright f\ a\ b \hookrightarrow f\ a\ b : C$. □

12.1.3 Basic change structures on types

After studying basic change structures in the abstract, we apply them to the study of our object language.

For each type τ, we can define a basic change structure on domain $[\![\tau]\!]$, which we write $\widetilde{\tau}$. Language plugins must provide basic change structures for base types. To provide basic change structures for function types $\sigma \to \tau$, we use the earlier construction for basic change structures $\widetilde{\sigma} \to \widetilde{\tau}$ on function spaces $[\![\sigma \to \tau]\!]$ (Definition 12.1.8).

Definition 12.1.15 (Basic change structures on types)
For each type τ we associate a basic change structure on domain $[\![\tau]\!]$, written $\widetilde{\tau}$ through the following equations:

$$\widetilde{\iota} = \dots$$
$$\widetilde{\sigma \to \tau} = \widetilde{\sigma} \to \widetilde{\tau}$$

Plugin Requirement 12.1.16 (Basic change structures on base types)
For each base type ι, the plugin defines a basic change structure on $[\![\iota]\!]$ that we write $\widetilde{\iota}$. □

Crucially, for each type τ we can define a type $\Delta\tau$ of changes, that we call *change type*, such that the change set $\Delta [\![\tau]\!]$ is just the domain $[\![\Delta\tau]\!]$ associated to change type $\Delta\tau$: $\Delta [\![\tau]\!] = [\![\Delta\tau]\!]$. This equation allows writing change terms that evaluate directly to change values.[3]

Definition 12.1.17 (Change types)
The change type $\Delta\tau$ of a type τ is defined as follows:

$$\Delta\iota = \dots$$
$$\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau$$

□

Lemma 12.1.18 (Δ and $[\![-]\!]$ commute on types)
For each type τ, change set $\Delta [\![\tau]\!]$ equals the domain of change type $[\![\Delta\tau]\!]$. □

Proof. By induction on types. For the case of function types, we simply prove equationally that $\Delta [\![\sigma \to \tau]\!] = \Delta([\![\sigma]\!] \to [\![\tau]\!]) = [\![\sigma]\!] \to \Delta [\![\sigma]\!] \to \Delta [\![\tau]\!] = [\![\sigma]\!] \to [\![\Delta\sigma]\!] \to [\![\Delta\tau]\!] = [\![\sigma \to \Delta\sigma \to \Delta\tau]\!] = [\![\Delta(\sigma \to \tau)]\!]$. The case for base types is delegates to plugins (Plugin Requirement 12.1.19). □

[3] Instead, in earlier proofs [Cai et al., 2014] the values of change terms were not change values, but had to be related to change values through a logical relation; see Sec. 15.4.

Plugin Requirement 12.1.19 (Base change types)
For each base type ι, the plugin defines a change type $\Delta\iota$ such that $\Delta [\![\iota]\!] = [\![\Delta\iota]\!]$. □

We refer to values of change types as *change values* or just *changes*.

Notation 12.1.20
We write basic change structures for types $\widetilde{\tau}$, not $\widetilde{[\![\tau]\!]}$, and $dv \triangleright v_1 \hookrightarrow v_2 : \tau$, not $dv \triangleright v_1 \hookrightarrow v_2 : [\![\tau]\!]$. We also write consistently $[\![\Delta\tau]\!]$, not $\Delta [\![\tau]\!]$. □

12.1.4 Validity as a logical relation

Next, we show an equivalent definition of validity for values of terms, directly by induction on types, as a ternary *logical* relation between a change, its source and destination. A typical logical relation constrains *functions* to map related input to related outputs. In a twist, validity constrains *function changes* to map related inputs to related outputs.

Definition 12.1.21 (Change validity)
We say that dv is a (valid) change from v_1 to v_2 (on type τ), and write $dv \triangleright v_1 \hookrightarrow v_2 : \tau$, if $dv : [\![\Delta\tau]\!]$, $v_1, v_2 : [\![\tau]\!]$ and dv is a "valid" description of the difference from v_1 to v_2, as we define in Fig. 12.1e. □

The key equations for function types are:

$$\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau$$
$$df \triangleright f_1 \hookrightarrow f_2 : \sigma \to \tau = \forall da \triangleright a_1 \hookrightarrow a_2 : \sigma.\ df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : \tau$$

Remark 12.1.22
We have kept repeating the idea that valid function changes map valid input changes to valid output changes. As seen in Sec. 10.4 and Lemmas 12.1.10 and 12.1.14, such valid outputs can in turn be valid function changes. We'll see the same idea at work in Lemma 12.1.27, in the correctness proof of $\mathcal{D}[\![-]\!]$.

As we have finally seen in this section, this definition of validity can be formalized as a logical relation, defined by induction on types. We'll later take for granted the consequences of validity, together with lemmas such as Lemma 12.1.10. □

12.1.5 Change structures on typing contexts

To describe changes to the inputs of a term, we now also introduce change contexts $\Delta\Gamma$, environment changes $d\rho : [\![\Delta\Gamma]\!]$, and validity for environment changes $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$.

A valid environment change from $\rho_1 : [\![\Gamma]\!]$ to $\rho_2 : [\![\Gamma]\!]$ is an environment $d\rho : [\![\Delta\Gamma]\!]$ that extends environment ρ_1 with valid changes for each entry. We first define the shape of environment changes through *change contexts*:

Definition 12.1.23 (Change contexts)
For each context Γ we define change context $\Delta\Gamma$ as follows:

$$\Delta\varepsilon = \varepsilon$$
$$\Delta(\Gamma, x : \tau) = \Delta\Gamma, x : \tau, dx : \Delta\tau.$$ □

Then, we describe validity of environment changes via a judgment.

Definition 12.1.24 (Environment change validity)
We define validity for environment changes through judgment $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$, pronounced "$d\rho$ is an environment change from ρ_1 to ρ_2 (at context Γ)", where $\rho_1, \rho_2 : [\![\Gamma]\!]$, $d\rho : [\![\Delta\Gamma]\!]$, via the following inference rules:

$$\frac{}{\varnothing \triangleright \varnothing \hookrightarrow \varnothing : \varepsilon} \qquad \frac{d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma \qquad da \triangleright a_1 \hookrightarrow a_2 : \tau}{(d\rho, x = a_1, dx = da) \triangleright (\rho_1, x = a_1) \hookrightarrow (\rho_2, x = a_2) : \Gamma, x : \tau}$$

□

Definition 12.1.25 (Basic change structures for contexts)
To each context Γ we associate a basic change structure on set $[\![\Gamma]\!]$. We take $[\![\Delta\Gamma]\!]$ as change set and reuse validity on environment changes (Definition 12.1.24). □

Notation 12.1.26
We write $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ rather than $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : [\![\Gamma]\!]$. □

Finally, to state and prove correctness of differentiation, we are going to need to discuss function changes on term semantics. The semantics of a term $\Gamma \vdash t : \tau$ is a function $[\![t]\!]$ from environments in $[\![\Gamma]\!]$ to values in $[\![\tau]\!]$. To discuss changes to $[\![t]\!]$ we need a basic change structure on function space $[\![\Gamma]\!] \to [\![\tau]\!]$.

Lemma 12.1.27
The construction of basic change structures on function spaces (Definition 12.1.8) associates to each context Γ and type τ a basic change structure $\widetilde{\Gamma} \to \widetilde{\tau}$ on function space $[\![\Gamma]\!] \to [\![\tau]\!]$. □

Notation 12.1.28
As usual, we write the change set as $\Delta([\![\Gamma]\!] \to [\![\tau]\!])$; for validity, we write $df \triangleright f_1 \hookrightarrow f_2 : \Gamma, \tau$ rather than $df \triangleright f_1 \hookrightarrow f_2 : [\![\Gamma]\!] \to [\![\tau]\!]$. □

12.2 Correctness of differentiation

In this section we state and prove correctness of differentiation, a term-to-term transformation written $\mathcal{D}[\![t]\!]$ that produces incremental programs. We recall that all our results apply only to well-typed terms (since we formalize no other ones).

Earlier, we described how $\mathcal{D}[\![-]\!]$ behaves through Slogan 10.3.3 — here is it again, for reference:

Slogan 10.3.3
Term $\mathcal{D}[\![t]\!]$ maps input changes to output changes. That is, $\mathcal{D}[\![t]\!]$ applied to initial base inputs and valid *input changes* (from initial inputs to updated inputs) gives a valid *output change* from t applied on old inputs to t applied on new inputs. □

In our slogan we do not specify what we meant by inputs, though we gave examples during the discussion. We have now the notions needed for a more precise statement. Term $\mathcal{D}[\![t]\!]$ must satisfy our slogan for two sorts of inputs:

1. Evaluating $\mathcal{D}[\![t]\!]$ must map an environment change $d\rho$ from ρ_1 to ρ_2 into a valid result change $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho$, going from $[\![t]\!]\ \rho_1$ to $[\![t]\!]\ \rho_2$.

2. As we learned since stating our slogan, validity is defined by recursion over types. If term t has type $\sigma \to \tau$, change $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ can in turn be a (valid) function change (Remark 12.1.22). Function changes map valid changes for *their* inputs to valid changes for *their* outputs.

Instead of saying that $[\![\mathcal{D}[\![t]\!]]\!]$ maps $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ to a change from $[\![t]\!]\ \rho_1$ to $[\![t]\!]\ \rho_2$, we can say that function $[\![t]\!]^\Delta = \lambda\rho\ d\rho \to [\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ must be a *nil change* for $[\![t]\!]$, that is, a *derivative* for $[\![t]\!]$. We give a name to this function change, and state $\mathcal{D}[\![-]\!]$'s correctness theorem.

Definition 12.2.1 (Incremental semantics)
We define the *incremental semantics* of a well-typed term $\Gamma \vdash t : \tau$ in terms of differentiation as:

$$[\![t]\!]^\Delta = (\lambda\rho_1\ d\rho \to [\![\mathcal{D}[\![t]\!]]\!]\ d\rho) : [\![\Gamma]\!] \to [\![\Delta\Gamma]\!] \to [\![\Delta\tau]\!]\,.$$ □

Theorem 12.2.2 ($\mathcal{D}[\![-]\!]$ is correct)
Function $[\![t]\!]^\Delta$ is a derivative of $[\![t]\!]$. That is, if $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \tau$. □

For now we discuss this statement further; we defer the proof to Sec. 12.2.2.

Remark 12.2.3 (Why $[\![-]\!]^\Delta$ ignores ρ_1)
Incremental semantics $[\![t]\!]^\Delta = \lambda\rho_1\ d\rho \to [\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ can safely ignore ρ_1 because $[\![-]\!]^\Delta$ assumes that change environment $d\rho$ is valid ($d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$), so $d\rho$ extends environment ρ_1 and ρ_1 provides no further information. □

Remark 12.2.4 (Term derivatives)
In Chapter 10, we suggested that $\mathcal{D}[\![t]\!]$ only produced a derivative for closed terms, not for open ones. But $[\![t]\!]^\Delta = \lambda\rho\ d\rho \to [\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ is *always* a nil change and derivative of $[\![t]\!]$ for any $\Gamma \vdash t : \tau$. There is no contradiction, because the *value* of $\mathcal{D}[\![t]\!]$ is $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho$, which is only a nil change if $d\rho$ is a nil change as well. In particular, for closed terms ($\Gamma = \varepsilon$), $d\rho$ must equal the empty environment $\emptyset$, hence $d\rho$ is a nil change. If τ is a function type, $df = [\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ accepts further inputs; since df must be a valid function change, it will also map them to valid outputs as required by our Slogan 10.3.3. Finally, if $\Gamma = \varepsilon$ and τ is a function type, then $df = [\![\mathcal{D}[\![t]\!]]\!]\ \emptyset$ is a derivative of $f = [\![t]\!]\ \emptyset$.

We summarize this remark with the following definition and corollary. □

Definition 12.2.5 (Derivatives of terms)
For all closed terms of function type $\vdash t : \sigma \to \tau$, we call $\mathcal{D}[\![t]\!]$ the (term) derivative of t. □

Corollary 12.2.6 (Term derivatives evaluate to derivatives)
For all closed terms of function type $\vdash t : \sigma \to \tau$, function $[\![\mathcal{D}[\![t]\!]]\!]\ \emptyset$ is a derivative of $[\![t]\!]\ \emptyset$. □

Proof. Because $[\![t]\!]^\Delta$ is a derivative (Theorem 12.2.2), and applying derivative $[\![t]\!]^\Delta$ to nil change $\emptyset$ gives a derivative (Lemma 12.1.13). □

Remark 12.2.7
We typically talk *a* derivative of a function value $f : A \to B$, not *the* derivative, since multiple different functions can satisfy the specification of derivatives. We talk about *the derivative* to refer to a canonically chosen derivative. For terms and their semantics, the canonical derivative the one produced by differentiation. For language primitives, the canonical derivative is the one chosen by the language plugin under consideration. □

Theorem 12.2.2 only makes sense if $\mathcal{D}[\![-]\!]$ has the right static semantics:

Lemma 12.2.8 (Typing of $\mathcal{D}[\![-]\!]$)
Typing rule

$$\frac{\Gamma \vdash t : \tau}{\Delta\Gamma \vdash \mathcal{D}[\![t]\!] : \Delta\tau}\ \textsc{Derive}$$

is derivable. □

After we'll define $\oplus$, in next chapter, we'll be able to relate $\oplus$ to validity, by proving Lemma 13.4.4, which we state in advance here:

Lemma 13.4.4 ($\oplus$ agrees with validity)
If $dv \triangleright v_1 \hookrightarrow v_2 : \tau$ then $v_1 \oplus dv = v_2$. □

Hence, updating base result $[\![t]\!]\ \rho_1$ by change $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho$ via $\oplus$ gives the updated result $[\![t]\!]\ \rho_2$.

Corollary 13.4.5 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
If $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![t]\!]\ \rho_1 \oplus [\![\mathcal{D}[\![t]\!]]\!]\ d\rho = [\![t]\!]\ \rho_2$. □

We anticipate the proof of this corollary:

Proof. First, differentiation is correct (Theorem 12.2.2), so under the hypotheses

$$[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \tau;$$

that judgement implies the thesis

$$[\![t]\!]\ \rho_1 \oplus [\![\mathcal{D}[\![t]\!]]\!]\ d\rho = [\![t]\!]\ \rho_2$$

because $\oplus$ agrees with validty (Lemma 13.4.4). □

12.2.1 Plugin requirements

Differentiation is extended by plugins on constants, so plugins must prove their extensions correct.

Plugin Requirement 12.2.9 (Typing of $\mathcal{D}^{\mathcal{C}}[\![-]\!]$)
For all $\vdash^{\mathcal{C}} c : \tau$, the plugin defines $\mathcal{D}^{\mathcal{C}}[\![c]\!]$ satisfying $\vdash \mathcal{D}^{\mathcal{C}}[\![c]\!] : \Delta\tau$. □

Plugin Requirement 12.2.10 (Correctness of $\mathcal{D}^{\mathcal{C}}[\![-]\!]$)
For all $\vdash^{\mathcal{C}} c : \tau$, $[\![\mathcal{D}^{\mathcal{C}}[\![c]\!]]\!]$ is a derivative for $[\![c]\!]$. □

Since constants are typed in the empty context, and the only change for an empty environment is an empty environment, Plugin Requirement 12.2.10 means that for all $\vdash^{\mathcal{C}} c : \tau$ we have

$$[\![\mathcal{D}^{\mathcal{C}}[\![c]\!]]\!]\ \varnothing \triangleright [\![c]\!]\ \varnothing \hookrightarrow [\![c]\!]\ \varnothing : \tau.$$

12.2.2 Correctness proof

We next recall $\mathcal{D}[\![-]\!]$'s definition and prove it satisfies its correctness statement Theorem 12.2.2.

Definition 10.5.1 (Differentiation)
Differentiation is the following term transformation:

$$\begin{aligned}\mathcal{D}[\![\, \lambda(x:\sigma) \to t \,]\!] &= \lambda(x:\sigma)\ (dx:\Delta\sigma) \to \mathcal{D}[\![\, t \,]\!] \\ \mathcal{D}[\![\, s\ t \,]\!] &= \mathcal{D}[\![\, s \,]\!]\ \ t\ \mathcal{D}[\![\, t \,]\!] \\ \mathcal{D}[\![\, x \,]\!] &= dx \\ \mathcal{D}[\![\, c \,]\!] &= \mathcal{D}^C[\![\, c \,]\!]\end{aligned}$$ □

where $\mathcal{D}^C[\![\, c \,]\!]$ defines differentiation on primitives and is provided by language plugins (see Appendix A.2.5), and dx stands for a variable generated by prefixing x's name with d, so that $\mathcal{D}[\![\, y \,]\!] = dy$ and so on.

Before correctness, we prove Lemma 12.2.8:

Lemma 12.2.8 (Typing of $\mathcal{D}[\![-]\!]$)
Typing rule

$$\frac{\Gamma \vdash t : \tau}{\Delta\Gamma \vdash \mathcal{D}[\![\, t \,]\!] : \Delta\tau}\ \textsc{Derive}$$

is derivable. □

Proof. The thesis can be proven by induction on the typing derivation $\Gamma \vdash t : \tau$. The case for constants is delegated to plugins in Plugin Requirement 12.2.9. □

We prove Theorem 12.2.2 using a typical logical relations strategy. We proceed by induction on term t and prove for each case that if $\mathcal{D}[\![-]\!]$ preserves validity on subterms of t, then also $\mathcal{D}[\![\, t \,]\!]$ preserves validity. Hence, if the input environment change $d\rho$ is valid, then the result of differentiation evaluates to valid change $[\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ \ d\rho$.

Readers familiar with logical relations proofs should be able to reproduce this proof on their own, as it is rather standard, once one uses the given definitions. In particular, this proof resembles closely the proof of the abstraction theorem or relational parametricity (as given by Wadler [1989, Sec. 6] or by Bernardy and Lasson [2011, Sec. 3.3, Theorem 3]) and the proof of the fundamental theorem of logical relations by Statman [1985].

Nevertheless, we spell this proof out, and use it to motivate how $\mathcal{D}[\![-]\!]$ is defined, more formally than we did in Sec. 10.5. For each case, we first give a short proof sketch, and then redo the proof in more detail to make the proof easier to follow.

Theorem 12.2.2 ($\mathcal{D}[\![-]\!]$ is correct)
Function $[\![\, t \,]\!]^\Delta$ is a derivative of $[\![\, t \,]\!]$. That is, if $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ \ d\rho \triangleright [\![\, t \,]\!]\ \rho_1 \hookrightarrow [\![\, t \,]\!]\ \rho_2 : \tau$. □

Proof. By induction on typing derivation $\Gamma \vdash t : \tau$.

- Case $\Gamma \vdash x : \tau$. The thesis is that $[\![\, \mathcal{D}[\![\, x \,]\!] \,]\!]$ is a derivative for $[\![\, x \,]\!]$, that is $[\![\, \mathcal{D}[\![\, x \,]\!] \,]\!]\ \ d\rho \triangleright [\![\, x \,]\!]\ \rho_1 \hookrightarrow [\![\, x \,]\!]\ \rho_2 : \tau$. Since $d\rho$ is a valid environment change from ρ_1 to ρ_2, $[\![\, dx \,]\!]\ d\rho$ is a valid change from $[\![\, x \,]\!]\ \rho_1$ to $[\![\, x \,]\!]\ \rho_2$. Hence, defining $\mathcal{D}[\![\, x \,]\!] = dx$ satisfies our thesis.
- Case $\Gamma \vdash s\ t : \tau$. The thesis is that $[\![\, \mathcal{D}[\![\, s\ t \,]\!] \,]\!]$ is a derivative for $[\![\, s\ t \,]\!]$, that is $[\![\, \mathcal{D}[\![\, s\ t \,]\!] \,]\!]\ \ d\rho \triangleright [\![\, s\ t \,]\!]\ \rho_1 \hookrightarrow [\![\, s\ t \,]\!]\ \rho_2 : \tau$. By inversion of typing, there is some type σ such that $\Gamma \vdash s : \sigma \to \tau$ and $\Gamma \vdash t : \sigma$.

To prove the thesis, in short, you can apply the inductive hypothesis to s and t, obtaining respectively that $[\![\mathcal{D}[\![s]\!]]\!]$ and $[\![\mathcal{D}[\![t]\!]]\!]$ are derivatives for $[\![s]\!]$ and $[\![t]\!]$. In particular, $[\![\mathcal{D}[\![s]\!]]\!]$ evaluates to a validity-preserving function change. Term $\mathcal{D}[\![s\ t]\!]$, that is $\mathcal{D}[\![s]\!]\ t\ \mathcal{D}[\![t]\!]$, applies validity-preserving function $\mathcal{D}[\![s]\!]$ to valid input change $\mathcal{D}[\![t]\!]$, and this produces a valid change for $s\ t$ as required.

In detail, our thesis is that for all $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ we have

$$[\![\mathcal{D}[\![s\ t]\!]]\!]\ d\rho \triangleright [\![s\ t]\!]\ \rho_1 \hookrightarrow [\![s\ t]\!]\ \rho_2 : \tau,$$

where $[\![s\ t]\!]\ \rho = ([\![s]\!]\ \rho)\ ([\![t]\!]\ \rho)$ and

$$\begin{aligned}
&[\![\mathcal{D}[\![s\ t]\!]]\!]\ d\rho \\
=\ &[\![\mathcal{D}[\![s]\!]\ t\ \mathcal{D}[\![t]\!]]\!]\ d\rho \\
=\ &([\![\mathcal{D}[\![s]\!]]\!]\ d\rho)\ ([\![t]\!]\ d\rho)\ ([\![\mathcal{D}[\![t]\!]]\!]\ d\rho) \\
=\ &([\![\mathcal{D}[\![s]\!]]\!]\ d\rho)\ ([\![t]\!]\ \rho_1)\ ([\![\mathcal{D}[\![t]\!]]\!]\ d\rho)
\end{aligned}$$

The last step relies on $[\![t]\!]\ d\rho = [\![t]\!]\ \rho_1$. Since weakening preserves meaning (Lemma A.2.8), this follows because $d\rho : [\![\Delta\Gamma]\!]$ extends $\rho_1 : [\![\Gamma]\!]$, and t can be typed in context Γ.

Our thesis becomes

$$[\![\mathcal{D}[\![s]\!]]\!]\ d\rho\ ([\![t]\!]\ \rho_1)\ ([\![\mathcal{D}[\![t]\!]]\!]\ d\rho) \triangleright [\![s]\!]\ \rho_1\ ([\![t]\!]\ \rho_1) \hookrightarrow [\![s]\!]\ \rho_2\ ([\![t]\!]\ \rho_2) : \tau.$$

By the inductive hypothesis on s and t we have

$$\begin{gathered}
[\![\mathcal{D}[\![s]\!]]\!]\ d\rho \triangleright [\![s]\!]\ \rho_1 \hookrightarrow [\![s]\!]\ \rho_2 : \sigma \to \tau \\
[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \sigma.
\end{gathered}$$

Since $[\![s]\!]$ is a function, its validity means

$$\forall da \triangleright a_1 \hookrightarrow a_2 : \sigma.\ [\![\mathcal{D}[\![s]\!]]\!]\ d\rho\ a_1\ da \triangleright [\![s]\!]\ \rho_1\ a_1 \hookrightarrow [\![s]\!]\ \rho_2\ a_2 : \tau.$$

Instantiating in this statement the hypothesis $da \triangleright a_1 \hookrightarrow a_2 : \sigma$ by $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \sigma$ gives the thesis.

- Case $\Gamma \vdash \lambda x \to t : \sigma \to \tau$. By inversion of typing, $\Gamma, x : \sigma \vdash t : \tau$. By typing of $\mathcal{D}[\![-]\!]$ you can show that

  $$\Delta\Gamma, x : \sigma, dx : \Delta\sigma \vdash \mathcal{D}[\![t]\!] : \Delta\tau.$$

  In short, our thesis is that $[\![\lambda x \to t]\!]^\Delta = \lambda\rho_1\ d\rho \to [\![\lambda x\ dx \to \mathcal{D}[\![t]\!]]\!]\ d\rho$ is a derivative of $[\![\lambda x \to t]\!]$. After a few simplifications, our thesis reduces to

  $$[\![\mathcal{D}[\![t]\!]]\!]\ (d\rho, x = a_1, dx = da) \triangleright [\![t]\!]\ (\rho_1, x = a_1) \hookrightarrow [\![t]\!]\ (\rho_2, x = a_2) : \tau$$

  for all $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ and $da \triangleright a_1 \hookrightarrow a_2 : \sigma$. But then, the thesis is simply that $[\![t]\!]^\Delta$ is the derivative of $[\![t]\!]$, which is true by inductive hypothesis.

  More in detail, our thesis is that $[\![\lambda x \to t]\!]^\Delta$ is a derivative for $[\![\lambda x \to t]\!]$, that is

  $$\begin{aligned}
  &\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma. \\
  &\qquad [\![\mathcal{D}[\![\lambda x \to t]\!]]\!]\ d\rho \triangleright [\![\lambda x \to t]\!]\ \rho_1 \hookrightarrow [\![\lambda x \to t]\!]\ \rho_2 : \sigma \to \tau \qquad (12.1)
  \end{aligned}$$

By simplifying, the thesis Eq. (12.1) becomes

$$\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma.$$
$$\lambda a_1\ da \to [\![\mathcal{D} [\![t]\!]]\!]\ (d\rho, x = a_1, dx = da) \triangleright$$
$$(\lambda a_1 \to [\![t]\!]\ (\rho_1, x = a_1)) \hookrightarrow (\lambda a_2 \to [\![t]\!]\ (\rho_2, x = a_2)) : \sigma \to \tau. \quad (12.2)$$

By definition of validity of function type, the thesis Eq. (12.2) becomes

$$\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma.\ \forall da \triangleright a_1 \hookrightarrow a_2 : \sigma.$$
$$[\![\mathcal{D} [\![t]\!]]\!]\ (d\rho, x = a_1, dx = da) \triangleright$$
$$[\![t]\!]\ (\rho_1, x = a_1) \hookrightarrow [\![t]\!]\ (\rho_2, x = a_2) : \tau. \quad (12.3)$$

To prove the rewritten thesis Eq. (12.3), take the inductive hypothesis on t: it says that $[\![\mathcal{D} [\![t]\!]]\!]$ is a derivative for $[\![t]\!]$, so $[\![\mathcal{D} [\![t]\!]]\!]$ maps valid environment changes on $\Gamma, x : \sigma$ to valid changes on τ. But by inversion of the validity judgment, all valid environment changes on $\Gamma, x : \sigma$ can be written as

$$(d\rho, x = a_1, dx = da) \triangleright (\rho_1, x = a_1) \hookrightarrow (\rho_2, x = a_2) : \Gamma, x : \sigma,$$

for valid changes $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ and $da \triangleright a_1 \hookrightarrow a_2 : \sigma$. So, the inductive hypothesis is that

$$\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma.\ \forall da \triangleright a_1 \hookrightarrow a_2 : \sigma.$$
$$[\![\mathcal{D} [\![t]\!]]\!]\ (d\rho, x = a_1, dx = da) \triangleright$$
$$[\![t]\!]\ (\rho_1, x = a_1) \hookrightarrow [\![t]\!]\ (\rho_2, x = a_2) : \tau. \quad (12.4)$$

But that is exactly our thesis Eq. (12.3), so we're done!

- Case $\Gamma \vdash c : \tau$. In essence, since weakening preserves meaning, we can rewrite the thesis to match Plugin Requirement 12.2.10.

  In more detail, the thesis is that $\mathcal{D}^C [\![c]\!]$ is a derivative for c, that is, if $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![\mathcal{D} [\![c]\!]]\!]\ d\rho \triangleright [\![c]\!]\ \rho_1 \hookrightarrow [\![c]\!]\ \rho_2 : \tau$. Since constants don't depend on the environment and weakening preserves meaning (Lemma A.2.8), and by the definitions of $[\![-]\!]$ and $\mathcal{D}[\![-]\!]$ on constants, the thesis can be simplified to $[\![\mathcal{D}^C [\![c]\!]]\!]\ \varnothing \triangleright [\![c]\!]\ \varnothing \hookrightarrow [\![c]\!]\ \varnothing : \tau$, which is delegated to plugins in Plugin Requirement 12.2.10. □

12.3 Discussion

12.3.1 The correctness statement

We might have asked for the following correctness property:

Theorem 12.3.1 (Incorrect correctness statement)
If $\Gamma \vdash t : \tau$ and $\rho_1 \oplus d\rho = \rho_2$ then $([\![t]\!]\ \rho_1) \oplus ([\![\mathcal{D} [\![t]\!]]\!]\ d\rho) = ([\![t]\!]\ \rho_2)$. □

However, this property is not quite right. We can only prove correctness if we restrict the statement to input changes $d\rho$ that are *valid*. Moreover, to prove this statement by induction we need to strengthen its conclusion: we require that the output change $[\![\mathcal{D} [\![t]\!]]\!]\ d\rho$ is also valid. To

see why, consider term $(\lambda x \to s)\ t$: Here the output of t is an input of s. Similarly, in $\mathcal{D}[\![\ (\lambda x \to s)\ t\ ]\!]$, the output of $\mathcal{D}[\![\ t\ ]\!]$ becomes an input change for subterm $\mathcal{D}[\![\ t\ ]\!]$, and $\mathcal{D}[\![\ s\ ]\!]$ behaves correctly only if only if $\mathcal{D}[\![\ t\ ]\!]$ produces a valid change.

Typically, change types contain values that invalid in some sense, but incremental programs will *preserve* validity. In particular, valid changes between functions are in turn functions that take valid input changes to valid output changes. This is why we formalize validity as a logical relation.

12.3.2 Invalid input changes

To see concretely why invalid changes, in general, can cause derivatives to produce incorrect results, consider again $grandTotal = \lambda xs\ ys \to sum\ (merge\ xs\ ys)$ from Sec. 10.2. Suppose a bag change dxs removes an element 20 from input bag xs, while dys makes no changes to ys: in this case, the output should decrease, so $dz = dgrandTotal\ xs\ dxs\ ys\ dys$ should be -20. However, that is only correct if 20 is actually an element of xs. Otherwise, $xs \oplus dxs$ will make no change to xs, hence the correct output change dz would be 0 instead of -20. Similar but trickier issues apply with function changes; see also Sec. 15.2.

12.3.3 Alternative environment changes

Environment changes can also be defined differently. We will use this alternative definition later (in Appendix D.2.2).

A change $d\rho$ from ρ_1 to ρ_2 contains a copy of ρ_1. Thanks to this copy, we can use an environment change as environment for the result of differentiation, that is, we can evaluate $\mathcal{D}[\![\ t\ ]\!]$ with environment $d\rho$, and Definition 12.2.1 can define $[\![\ t\ ]\!]^\Delta$ as $\lambda\rho_1\ d\rho \to [\![\ \mathcal{D}[\![\ t\ ]\!]\ ]\!]\ d\rho$.

But we could adapt definitions to omit the copy of ρ_1 from $d\rho$, by setting

$$\Delta\,(\Gamma, x : \tau) = \Delta\Gamma, dx : \Delta\tau$$

and adapting other definitions. Evaluating $\mathcal{D}[\![\ t\ ]\!]$ still requires base inputs; we could then set $[\![\ t\ ]\!]^\Delta = \lambda\rho_1\ d\rho \to [\![\ \mathcal{D}[\![\ t\ ]\!]\ ]\!]\ (\rho_1, d\rho)$, where $\rho_1, d\rho$ simply merges the two environments appropriately (we omit a formal definition). This is the approach taken by Cai et al. [2014]. When proving Theorem 12.2.2, using one or the other definition for environment changes makes little difference; if we embed the base environment in environment changes, we reduce noise because we need not define environment meging formally.

Later (in Appendix D.2.2) we will deal with environment explicitly, and manipulate them in programs. Then we will use this alternative definition for environment changes, since it will be convenient to store base environments separately from environment changes.

12.3.4 Capture avoidance

Differentiation generates new names, so a correct implementation must prevent accidental capture. Till now we have ignored this problem.

Using de Bruijn indexes Our mechanization has no capture issues because it uses de Bruijn indexes. Change context just alternate variables for base inputs and input changes. A context such as $\Gamma = x : \mathbb{Z}, y : Bool$ is encoded as $\Gamma = \mathbb{Z}, Bool$; its change context is $\Delta\Gamma = \mathbb{Z}, \Delta\mathbb{Z}, Bool, \Delta Bool$. This solution is correct and robust, and is the one we rely on.

Alternatively, we can mechanize ILC using separate syntax for change terms dt that use separate namespaces for base variables and change variables.

$$
\begin{aligned}
ds, dt ::= {} & dc \\
\mid {} & \lambda(x : \sigma)\ (dx : \Delta\sigma) \to dt \\
\mid {} & ds\ t\ dt \\
\mid {} & dx
\end{aligned}
$$

In that case, change variables live in a separate namespace. Example context $\Gamma = \mathbb{Z}, \mathit{Bool}$ gives rise to a different sort of change context, $\Delta\Gamma = \Delta\mathbb{Z}, \Delta\mathit{Bool}$. And a change term in context Γ is evaluted with separate environments for Γ and $\Delta\Gamma$. This is appealing, because it allows defining differentiation and proving it correct without using weakening and applying its proof of soundness. We still need to use weakening to convert change terms to their equivalents in the base language, but proving that conversion correct is more straightforward.

Using names Next, we discuss issues in implementing this transformation with names rather than de Bruijn indexes. Using names rather than de Bruijn indexes makes terms more readable; this is also why in this thesis we use names in our on-paper formalization.

Unlike the rest of this chapter, we keep this discussion informal, also because we have not mechanized any definitions using names (as it may be possible using nominal logic), nor attempted formal proofs. The rest of the thesis does not depend on this material, so readers might want to skip to next section.

Using names introduces the risk of capture, as it is common for name-generating transformations [Erdweg et al., 2014]. For instance, differentiating term $t = \lambda x \to f\ dx$ gives $\mathcal{D}[\![\, t \,]\!] = \lambda x\ dx \to df\ dx\ ddx$. Here, variable dx represents a base input and is free in t, yet it is incorrectly captured in $\mathcal{D}[\![\, t \,]\!]$ by the other variable dx, the one representing x's change. Differentiation gives instead a correct result if we α-rename x in t to any other name (more on that in a moment).

A few workarounds and fixes are possible.

- As a workaround, we can forbid names starting with the letter d for variables in base terms, as we do in our examples; that's formally correct but pretty unsatisfactory and inelegant. With this approach, our term $t = \lambda x \to f\ dx$ is simply forbidden.

- As a better workaround, instead of prefixing variable names with d, we can add change variables as a separate construct to the syntax of variables and forbid differentiation on terms that containing change variables. This is a variant of the earlier approach using separate change terms. While we used this approach in our prototype implementation in Scala [Cai et al., 2014], it makes our output language annoyingly non-standard. Converting to a standard language using names (not de Bruijn indexes) raises again capture issues.

- We can try to α-rename *existing* bound variables, as in the implementation of capture-avoiding substitution. As mentioned, in our case, we can rename bound variable x to y and get $t' = \lambda y \to f\ dx$. Differentiation gives the correct result $\mathcal{D}[\![\, t' \,]\!] = \lambda y\ dy \to df\ dx\ ddx$. In general we can define $\mathcal{D}[\![\, \lambda x \to t \,]\!] = \lambda y\ dy \to \mathcal{D}[\![\, t\,[x := y] \,]\!]$ where neither y nor dy appears free in t; that is, we search for a fresh variable y (which, being fresh, does not appear anywhere else) such that also dy does not appear free in t.

  This solution is however subtle: it reuses ideas from capture-avoiding substitution, which is well-known to be subtle. We have not attempted to formally prove such a solution correct (or even test it) and have no plan to do so.

- Finally and most easily we can α-rename *new* bound variables, the ones used to refer to changes, or rather, only pick them fresh. But if we pick, say, fresh variable dx_1 to refer to the change of variable x, we *must* use dx_1 consistently for every occurrence of x, so that

$\mathcal{D}[\![\, \lambda x \to x \,]\!]$ is not $\lambda dx_1 \to dx_2$. Hence, we extend $\mathcal{D}[\![-]\!]$ to also take a map from names to names as follows:

$$\begin{aligned}
\mathcal{D}[\![\, \lambda(x : \sigma) \to t, m \,]\!] &= \lambda(x : \sigma)\ (dx : \Delta\sigma) \to \mathcal{D}[\![\, t, (m\,[x \to dx]) \,]\!] \\
\mathcal{D}[\![\, s\ t, m \,]\!] &= \mathcal{D}[\![\, s, m \,]\!]\ \ t\ \mathcal{D}[\![\, t, m \,]\!] \\
\mathcal{D}[\![\, x, m \,]\!] &= m(x) \\
\mathcal{D}[\![\, c, m \,]\!] &= \mathcal{D}^{\mathcal{C}}[\![\, c \,]\!]
\end{aligned}$$

where $m(x)$ represents lookup of x in map m, dx is now a fresh variable that does not appear in t, and $m\,[x \to dx]$ extend m with a new mapping from x to dx.

But this approach, that is using a map from base variables to change variables, affects the interface of differentiation. In particular, it affects which variables are free in output terms, hence we must also update the definition of $\Delta\Gamma$ and derived typing rule DERIVE. With this approach, if term s is closed then $\mathcal{D}[\![\, s, \mathit{emptyMap} \,]\!]$ gives a result α-equivalent to the old $\mathcal{D}[\![\, s \,]\!]$, as long as s triggers no capture issues. But if instead s is open, invoking $\mathcal{D}[\![\, s, \mathit{emptyMap} \,]\!]$ is not meaningful: we must pass an initial map m containing mappings from s's free variables to fresh variables for their changes. These change variables appear free in $\mathcal{D}[\![\, s, m \,]\!]$, hence we must update typing rule DERIVE, and modify $\Delta\Gamma$ to use m.

We define $\Delta_m\Gamma$ by adding m as a parameter to $\Delta\Gamma$, and use it in a modified rule DERIVE':

$$\begin{aligned}
\Delta_m\varepsilon &= \varepsilon \\
\Delta_m\,(\Gamma, x : \tau) &= \Delta_m\Gamma, x : \tau, m(x) : \Delta\tau.
\end{aligned}$$

$$\frac{\Gamma \vdash t : \tau}{\Delta_m\Gamma \vdash \mathcal{D}[\![\, t, m \,]\!] : \Delta\tau}\ \textsc{Derive'}$$

We conjecture that DERIVE' holds and that $\mathcal{D}[\![\, t, m \,]\!]$ is correct, but we have attempted no formal proof.

12.4 Plugin requirement summary

For reference, we repeat here plugin requirements spread through the chapter.

Plugin Requirement 12.1.16 (Basic change structures on base types)
For each base type ι, the plugin defines a basic change structure on $[\![\, \iota \,]\!]$ that we write $\widetilde{\iota}$. □

Plugin Requirement 12.1.19 (Base change types)
For each base type ι, the plugin defines a change type $\Delta\iota$ such that $\Delta\,[\![\, \iota \,]\!] = [\![\, \Delta\iota \,]\!]$. □

Plugin Requirement 12.2.9 (Typing of $\mathcal{D}^{\mathcal{C}}[\![-]\!]$)
For all $\vdash^{\mathcal{C}} c : \tau$, the plugin defines $\mathcal{D}^{\mathcal{C}}[\![\, c \,]\!]$ satisfying $\vdash \mathcal{D}^{\mathcal{C}}[\![\, c \,]\!] : \Delta\tau$. □

Plugin Requirement 12.2.10 (Correctness of $\mathcal{D}^{\mathcal{C}}[\![-]\!]$)
For all $\vdash^{\mathcal{C}} c : \tau$, $\left[\!\left[\, \mathcal{D}^{\mathcal{C}}[\![\, c \,]\!] \,\right]\!\right]$ is a derivative for $[\![\, c \,]\!]$. □

12.5 Chapter conclusion

In this chapter, we have formally defined changes for values and environments of our language, and when changes are valid. Through these definitions, we have explained that $\mathcal{D}[\![t]\!]$ is correct, that is, that it maps changes to the input environment to changes to the output environment. All of this assumes, among other things, that language plugins define valid changes for their base types and derivatives for their primitives.

In next chapter we discuss operations we provide to construct and use changes. These operations will be especially useful to provide derivatives of primitives.

Chapter 13

Change structures

In the previous chapter, we have shown that evaluating the result of differentiation produces a valid change dv from the old output v_1 to the new one v_2. To *compute* v_2 from v_1 and dv, in this chapter we introduce formally the operator $\oplus$ mentioned earlier.

To define differentiation on primitives, plugins need a few operations on changes, not just $\oplus$, $\ominus$, $\odot$ and $\mathbf{0}$.

To formalize these operators and specify their behavior, we extend the notion of basic change structure into the notion of *change structure* in Sec. 13.1. The change structure for function spaces is not entirely intuitive, so we motivate it in Sec. 13.2. Then, we show how to take change structures on A and B and define new ones on $A \to B$ in Sec. 13.3. Using these structures, we finally show that starting from change structures for base types, we define change structures for all types τ and contexts Γ in Sec. 13.4, completing the core theory of changes.

13.1 Formalizing $\oplus$ and change structures

In this section, we define what is a *change structure* on a set V. A change structure $\widehat{V}$ extends a basic change structure $\widetilde{V}$ with *change operators* $\oplus$, $\ominus$, $\odot$ and $\mathbf{0}$. Change structures also require change operators to respect validity, as described below. Key properties of change structures follow in Sec. 13.1.2.

As usual, we'll use metavariables v, v_1, v_2, ... will range over elements of V, while dv, dv_1, dv_2, ... will range over elements of ΔV.

Let's first recall change operators. Operator $\oplus$ ("oplus") updates a value with a change: If dv is a valid change from v_1 to v_2, then $v_1 \oplus dv$ (read as "v_1 updated by dv" or "v_1 oplus dv") is guaranteed to return v_2. Operator $\ominus$ ("ominus") produces a difference between two values: $v_2 \ominus v_1$ is a valid change from v_1 to v_2. Operator $\mathbf{0}$ ("nil") produces nil changes: $\mathbf{0}_v$ is a nil change for v. Finally, change composition $\odot$ ("ocompose") "pastes changes together": if dv_1 is a valid change from v_1 to v_2 and dv_2 is a valid change from v_2 to v_3, then $dv_1 \odot dv_2$ (read "dv_1 composed with dv_2") is a valid change from v_1 to v_3.

We summarize these descriptions in the following definition.

Definition 13.1.1
A change structure $\widehat{V}$ over a set V requires:

(a) A basic change structure for V (hence change set ΔV and validity $dv \triangleright v_1 \hookrightarrow v_2 : V$).

(b) An update operation $\oplus : V \to \Delta V \to V$ that *updates* a value with a change. Update must agree with validity: for all $dv \triangleright v_1 \hookrightarrow v_2 : V$ we require $v_1 \oplus dv = v_2$.

(c) A nil change operation $\mathbf{0} : V \to \Delta V$, that must produce nil changes: for all $v : V$ we require $\mathbf{0}_v \triangleright v \hookrightarrow v : V$.

(d) a difference operation $\ominus : V \to V \to \Delta V$ that produces a change across two values: for all $v_1, v_2 : V$ we require $v_2 \ominus v_1 \triangleright v_1 \hookrightarrow v_2 : V$.

(e) a change composition operation $\circledcirc : \Delta V \to \Delta V \to \Delta V$, that composes together two changes relative to a base value. Change composition must preserve validity: for all $dv_1 \triangleright v_1 \hookrightarrow v_2 : V$ and $dv_2 \triangleright v_2 \hookrightarrow v_3 : V$ we require $dv_1 \circledcirc dv_2 \triangleright v_1 \hookrightarrow v_3 : V$. □

Notation 13.1.2
Operators $\oplus$ and $\ominus$ can be subscripted to highlight their base set, but we will usually omit such subscripts. Moreover, $\oplus$ is left-associative, so that $v \oplus dv_1 \oplus dv_2$ means $(v \oplus dv_1) \oplus dv_2$.

Finally, whenever we have a change structure such as $\widehat{A}$, $\widehat{B}$, $\widehat{V}$, and so on, we write respectively A, B, V to refer to its base set. □

13.1.1 Example: Group changes

As an example, we show next that each group induces a change structure on its carrier. This change structure also subsumes basic change structures we saw earlier on integers.

Definition 13.1.3 (Change structure on groups G)
Given any group $(G, e, +, -)$ we can define a change structure $\widehat{G}$ on carrier set G as follows.

(a) The change set is G.

(b) Validity is defined as $dg \triangleright g_1 \hookrightarrow g_2 : G$ if and only if $g_2 = g_1 + dg$.

(c) Change update coincides with $+$: $g_1 \oplus dg = g_1 + dg$. Hence $\oplus$ agrees *perfectly* with validity: for all $g_1 \in G$, all changes dg are valid from g_1 to $g_1 \oplus dg$ (that is $dg \triangleright g_1 \hookrightarrow g_1 \oplus dg : G$).

(d) We define difference as $g_2 \ominus g_1 = (-g_1) + g_2$. Verifying $g_2 \ominus g_1 \triangleright g_1 \hookrightarrow g_2 : G$ reduces to verifying $g_1 + ((-g_1) + g_2) = g_2$, which follows from group axioms.

(e) The only nil change is the identity element: $\mathbf{0}_g = e$. Validity $\mathbf{0}_g \triangleright g \hookrightarrow g : G$ reduces to $g + e = g$ which follows from group axioms.

(f) Change composition also coincides with $+$: $dg_1 \circledcirc dg_2 = dg_1 + dg_2$. Let's prove that composition respects validity. Our hypothesis is $dg_1 \triangleright g_1 \hookrightarrow g_2 : G$ and $dg_2 \triangleright g_2 \hookrightarrow g_3 : G$. Because $\oplus$ agrees perfectly with validity, the hypothesis is equivalent to $g_2 = g_1 \oplus dg_1$ and

$$g_3 = g_2 \oplus dg_2 = (g_1 \oplus dg_1) \oplus dg_2.$$

Our thesis is $dg_1 \circledcirc dg_2 \triangleright g_1 \hookrightarrow g_3 : G$, that is

$$g_3 = g_1 \oplus (dg_1 \circledcirc dg_2).$$

Hence the thesis reduces to

$$(g_1 \oplus dg_1) \oplus dg_2 = g_1 \oplus (dg_1 \circledcirc dg_2),$$

hence to $g_1 + (dg_1 + dg_2) = (g_1 + dg_1) + dg_2$, which is just the associative law for group G. □

As we show later, group change structures are useful to support aggregation.

13.1.2 Properties of change structures

To understand better the definition of change structures, we present next a few lemmas following from this definition.

Lemma 13.1.4 ($\ominus$ inverts $\oplus$)
$\oplus$ inverts $\ominus$, that is

$$v_1 \oplus (v_2 \ominus v_1) = v_2,$$

for change structure $\widehat{V}$ and any values $v_1, v_2 : V$. □

Proof. For change structures, we know $v_2 \ominus v_1 \triangleright v_1 \hookrightarrow v_2 : V$, and $v_1 \oplus (v_2 \ominus v_1) = v_2$ follows.

More in detail: Change $dv = v_2 \ominus v_1$ is a valid change from v_1 to v_2 (because $\ominus$ produces valid changes, $v_2 \ominus v_1 \triangleright v_1 \hookrightarrow v_2 : V$), so updating dv's source v_1 with dv produces dv's destination v_2 (because $\oplus$ agrees with validity, that is if $dv \triangleright v_1 \hookrightarrow v_2 : V$ then $v_1 \oplus dv = v_2$). □

Lemma 13.1.5 (A change can't be valid for two destinations with the same source)
Given a change $dv : \Delta V$ and a source $v_1 : V$, dv can only be valid with v_1 as source for a *single* destination. That is, if $dv \triangleright v_1 \hookrightarrow v_{2a} : V$ and $dv \triangleright v_1 \hookrightarrow v_{2b} : V$ then $v_{2a} = v_{2b}$. □

Proof. The proof follows, intuitively, because $\oplus$ also maps change dv and its source v_1 to its destination, and $\oplus$ is a function.

More technically, since $\oplus$ respects validity, the hypotheses mean that $v_{2a} = v_1 \oplus dv = v_{2b}$ as required. □

Beware that, changes can be valid for multiple sources, and associate them to different destination. For instance, integer 0 is a valid change for all integers.

If a change dv has source v, dv's destination equals $v \oplus dv$. So, to specify that dv is valid with source v, without mentioning dv's destination, we introduce the following definition.

Definition 13.1.6 (One-sided validity)
We define relation $\mathcal{V}_A$ as $\{(v, dv) \in A \times \Delta A \mid dv \triangleright v \hookrightarrow v \oplus dv : V\}$. □

We use this definition right away:

Lemma 13.1.7 ($\odot$ and $\oplus$ interact correctly)
If $(v_1, dv_1) \in V_V$ and $(v_1 \oplus dv_1, dv_2) \in V_V$ then $v_1 \oplus (dv_1 \odot dv_2) = v_1 \oplus dv_1 \oplus dv_2$. □

Proof. We know that $\odot$ preserves validity, so under the hypotheses $(v_1, dv_1) \in V_V$ and $(v_1 \oplus dv_1, dv_2) \in V_V$ we get that $dv = dv_1 \odot dv_2$ is a valid change from v_1 to $v_1 \oplus dv_1 \oplus dv_2$:

$$dv_1 \odot dv_2 \triangleright v_1 \hookrightarrow v_1 : V \oplus dv_1 \oplus dv_2.$$

Hence, updating dv's source v_1 with dv produces dv's destination $v_1 \oplus dv_1 \oplus dv_2$:

$$v_1 \oplus (dv_1 \odot dv_2) = v_1 \oplus dv_1 \oplus dv_2. \qquad \text{fl}$$

We can define **0** in terms of other operations, and prove they satisfy their requirements for change structures.

Lemma 13.1.8 (0 can be derived from $\ominus$)
If we define $\mathbf{0}_v = v \ominus v$, then **0** produces valid changes as required ($\mathbf{0}_v \triangleright v \hookrightarrow v : V$), for any change structure $\widehat{V}$ and value $v : V$. □

Proof. This follows from validity of $\ominus$ ($v_2 \ominus v_1 \triangleright v_1 \hookrightarrow v_2 : V$) instantiated with $v_1 = v$ and $v_2 = v$.□

Moreover, nil changes are a right identity element for $\oplus$:

Corollary 13.1.9 (Nil changes are identity elements)
Any nil change dv for a value v is a right identity element for $\oplus$, that is we have $v \oplus dv = v$ for every set V with a basic change structure, every element $v \in V$ and every nil change dv for v. □

Proof. This follows from Lemma 13.4.4 and the definition of nil changes. □

The converse of this theorem does not hold: there exists changes dv such that $v \oplus dv = v$ yet dv is not a valid change from v to v. More in general, $v_1 \oplus dv = v_2$ does not imply that dv is a valid change. For instance, under earlier definitions for changes on naturals, if we take $v = 0$ and $dv = -5$, we have $v \oplus dv = v$ even though dv is not valid; examples of invalid changes on functions are discussed at Sec. 12.3.2 and Sec. 15.2. However, we prefer to define "dv is a nil change" as we do, to imply that dv is valid, not just a neutral element.

13.2 Operations on function changes, informally

13.2.1 Examples of nil changes

We have not defined any change structure yet, but we can already exhibit nil changes for some values, including a few functions.

Example 13.2.1

- An environment change for an empty environment $\varnothing : [\![\varepsilon]\!]$ must be an environment for the empty context $\Delta\varepsilon = \varepsilon$, so it must be the empty environment! In other words, if and only if $d\rho \triangleright \varnothing \hookrightarrow \varnothing : \varepsilon$, then and only then $d\rho = \varnothing$ and, in particular, $d\rho$ is a nil change.

- If all values in an environment ρ have nil changes, the environment has a nil change $d\rho = \mathbf{0}_\rho$ associating each value to a nil change. Indeed, take a context Γ and a suitable environment $\rho : [\![\Gamma]\!]$. For each typing assumption $x : \tau$ in Γ, assume we have a nil change $\mathbf{0}_v$ for v. Then we define $\mathbf{0}_\rho : [\![\Delta\Gamma]\!]$ by structural recursion on ρ as:

  $$\mathbf{0}_\varnothing = \varnothing$$
  $$\mathbf{0}_{\rho, x=v} = \mathbf{0}_\rho, x = v, dx = \mathbf{0}_v$$

  Then we can see that $\mathbf{0}_\rho$ is indeed a nil change for ρ, that is, $\mathbf{0}_\rho \triangleright \rho \hookrightarrow \rho : \Gamma$.

- We have seen in Theorem 12.2.2 that, whenever $\Gamma \vdash t : \tau$, $[\![t]\!]$ has nil change $[\![t]\!]^\Delta$. Moreover, if we have an appropriate nil environment change $d\rho \triangleright \rho \hookrightarrow \rho : \Gamma$ (which we often do, as discussed above), then we also get a nil change $[\![t]\!]^\Delta\ \rho\ d\rho$ for $[\![t]\!]\ \rho$:

  $$[\![t]\!]^\Delta\ \rho\ d\rho \triangleright [\![t]\!]\ \rho \hookrightarrow [\![t]\!]\ \rho : \tau.$$

  In particular, for all closed well-typed terms $\vdash t : \tau$ we have

  $$[\![t]\!]^\Delta\ \varnothing\ \varnothing \triangleright [\![t]\!]\ \varnothing \hookrightarrow [\![t]\!]\ \varnothing : \tau.$$

□

13.2.2 Nil changes on arbitrary functions

We have discussed how to find a nil change $\mathbf{0}_f$ for a function f if we know the *intension* of f, that is, its definition. What if we have only its *extension*, that is, its behavior? Can we still find $\mathbf{0}_f$? That's necessary to implement $\mathbf{0}$ as an object-language function $\mathbf{0}$ from f to $\mathbf{0}_f$, since such a function does not have access to f's implementation. That's also necessary to define $\mathbf{0}$ on metalanguage function spaces — and we look at this case first.

We seek a nil change $\mathbf{0}_f$ for an arbitrary metalanguage function $f : A \to B$, where A and B are arbitrary sets; we assume a basic change structure on $A \to B$, and will require A and B to support a few additional operations. We require that

$$\mathbf{0}_f \triangleright f \hookrightarrow f : A \to B. \tag{13.1}$$

Equivalently, whenever $da \triangleright a_1 \hookrightarrow a_2 : A$ then $\mathbf{0}_f\ a_1\ da \triangleright f\ a_1 \hookrightarrow f\ a_2 : B$. By Lemma 13.4.4, it follows that

$$f\ a_1 \oplus \mathbf{0}_f\ a_1\ da = f\ a_2, \tag{13.2}$$

where $a_1 \oplus da = a_2$.

To define $\mathbf{0}_f$ we solve Eq. (13.2). To understand how, we use an analogy. $\oplus$ and $\ominus$ are intended to resemble $+$ and $-$. To solve $f\ a_1 + \mathbf{0}_f\ a_1\ da = f\ a_2$, we subtract $f\ a_1$ from both sides and write $\mathbf{0}_f\ a_1\ da = f\ a_2 - f\ a_1$.

Similarly, here we use operator $\ominus$: $\mathbf{0}_f$ must equal

$$\mathbf{0}_f = \lambda a_1\ da \to f\ (a_1 \oplus da) \ominus f\ a_1. \tag{13.3}$$

Because $b_2 \ominus b_1 \triangleright b_1 \hookrightarrow b_2 : B$ for all $b_1, b_2 : B$, we can verify that $\mathbf{0}_f$ as defined by Eq. (13.3) satisfies our original requirement Eq. (13.1), not just its weaker consequence Eq. (13.2).

We have shown that, to define $\mathbf{0}$ on functions $f : A \to B$, we can use $\ominus$ at type B. Without using f's intension, we are aware of no alternative. To ensure $\mathbf{0}_f$ is defined for all f, we require that change structures define $\ominus$. We can then define $\mathbf{0}$ as a derived operation via $\mathbf{0}_v = v \ominus v$, and verify this derived definition satisfies requirements for $\mathbf{0}_{\_}$.

Next, we show how to define $\ominus$ on functions. We seek a valid function change $f_2 \ominus f_1$ from f_1 to f_2. We have just sought and found a valid change from f to f; generalizing the reasoning we used, we obtain that whenever $da \triangleright a_1 \hookrightarrow a_2 : A$ then we need to have $(f_2 \ominus f_1)\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B$; since $a_2 = a_1 \oplus da$, we can define

$$f_2 \ominus f_1 = \lambda a_1\ da \to f_2\ (a_1 \oplus da) \ominus f_1\ a_1. \tag{13.4}$$

One can verify that Eq. (13.4) defines $f_2 \ominus f_1$ as a valid function from f_1 to f_2, as desired. And after defining $f_2 \ominus f_1$, we need no more to define $\mathbf{0}_f$ separately using Eq. (13.3). We can just define $\mathbf{0}_f = f \ominus f$ simplify through the definition of $\ominus$ in Eq. (13.4), and reobtain Eq. (13.3) as a derived equation:

$$\mathbf{0}_f = f \ominus f = \lambda a_1\ da \to f\ (a_1 \oplus da) \ominus f\ a_1,$$

We defined $f_2 \ominus f_1$ on metalanguage functions. We can also internalize change operators in our object language, that is, define for each type τ object-level terms $\oplus_\tau$, $\ominus_\tau$, and so on, with the same behavior. We can define object-language change operators such as $\ominus$ using the same equations. However, the produced function change df is slow, because it recomputes the old output $f_1\ a_1$, computes the new output $f_2\ a_2$ and takes the difference.

However, we can implement $\ominus_{\sigma \to \tau}$ using replacement changes, if they are supported by the change structure on type τ. Let us define $\ominus_\tau$ as $b_2 \ominus b_1 = !b_2$ and simplify Eq. (13.4); we obtain

$$f_2 \ominus f_1 = \lambda a_1\ da \to !(f_2\ (a_1 \oplus da)).$$

We could even imagine allowing replacement changes on functions themselves. However, here the bang operator needs to be defined to produce a function change that can be applied, hence

$$!f_2 = \lambda a_1\ da \to !(f_2\ (a_1 \oplus da)).$$

Alternatively, as we see in Appendix D, we could represent function changes not as functions but as data through *defunctionalization*, and provide a function applying defunctionalized function changes $df : \Delta(\sigma \to \tau)$ to inputs $t_1 : \sigma$ and $dt : \Delta\sigma$. In this case, $!f_2$ would simply be another way to produce defunctionalized function changes.

13.2.3 Constraining $\oplus$ on functions

Next, we discuss how $\oplus$ must be defined on functions, and show informally why we must define $f_1 \oplus df = \lambda v \to f_1\ x \oplus df\ v\ \mathbf{0}_v$ to prove that $\oplus$ on functions agrees with validity (that is, Lemma 13.4.4).

We know that a valid function change $df \triangleright f_1 \hookrightarrow f_2 : A \to B$ takes valid input changes $dv \triangleright v_1 \hookrightarrow v_2 : A$ to a valid output change $df\ v_1\ dv \triangleright f_1\ v_1 \hookrightarrow f_2\ v_2 : B$. We require that $\oplus$ agrees with validity (Lemma 13.4.4), so $f_2 = f_1 \oplus df$, $v_2 = v_1 \oplus dv$ and

$$f_2\ v_2 = (f_1 \oplus df)\ (v_1 \oplus dv) = f_1\ v_1 \oplus df\ v_1\ dv. \tag{13.5}$$

Instantiating dv with $\mathbf{0}_v$ gives equation

$$(f_1 \oplus df)\ v_1 = (f_1 \oplus df)\ (v_1 \oplus \mathbf{0}_v) = f_1\ v_1 \oplus df\ v_1\ \mathbf{0}_v,$$

which is not only a requirement on $\oplus$ for functions but also defines $\oplus$ effectively.

13.3 Families of change structures

In this section, we derive change structures for $A \to B$ and $A \times B$ from two change structures $\widehat{A}$ and $\widehat{B}$. The change structure on $A \to B$ enables defining change structures for function types. Similarly, the change structure on $A \times B$ enables defining a change structure for product types in a language plugin, as described informally in Sec. 11.5. In Sec. 11.6 we also discuss informally change structures for disjoint sums: Formally, we can derive a change structure for disjoint union of sets $A + B$ (from change structures for A and B), and this enables defining change structures for sum types; we have mechanized the required proofs, but omit the tedious details here.

13.3.1 Change structures for function spaces

Sec. 13.2 introduces informally how to define change operations on $A \to B$. Next, we define formally change structures on function spaces, and then prove its operations respect their constraints.

Definition 13.3.1 (Change structure for $A \to B$)
Given change structures $\widehat{A}$ and $\widehat{B}$ we define a change structure on their function space $A \to B$, written $\widehat{A} \to \widehat{B}$, where:

(a) The change set is defined as: $\Delta(A \to B) = A \to \Delta A \to \Delta B$.

(b) Validity is defined as

$$df \triangleright f_1 \hookrightarrow f_2 : A \to B = \forall a_1\ da\ a_2.\ (da \triangleright a_1 \hookrightarrow a_2 : A) \\ \text{implies } (df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B).$$

(c) We define change update by

$$f_1 \oplus df = \lambda a \to f_1\ a \oplus df\ a\ \mathbf{0}_a.$$

(d) We define difference by

$$f_2 \ominus f_1 = \lambda a\ da \to f_2\ (a \oplus da) \ominus f_1\ a.$$

(e) We define $\mathbf{0}$ like in Lemma 13.1.8 as

$$\mathbf{0}_f = f \ominus f.$$

(f) We define change composition as

$$df_1 \circledcirc df_2 = \lambda a\ da \to df_1\ a\ \mathbf{0}_a \circledcirc df_2\ a\ da.$$ □

Lemma 13.3.2
Definition 13.3.1 defines a correct change structure $\widehat{A \to B}$. □

Proof.

- We prove that $\oplus$ agrees with validity on $A \to B$. Consider $f_1, f_2 : A \to B$ and $df \triangleright f_1 \hookrightarrow f_2 : A \to B$; we must show that $f_1 \oplus df = f_2$. By functional extensionality, we only need prove that $(f_1 \oplus df)\ a = f_2\ a$, that is that $f_1\ a \oplus df\ a\ \mathbf{0}_a = f_2\ a$. Since $\oplus$ agrees with validity on B, we just need to show that $df\ a\ \mathbf{0}_a \triangleright f_1\ a \hookrightarrow f_2\ a : B$, which follows because $\mathbf{0}_a$ is a valid change from a to a and because df is a valid change from f_1 to f_2.

- We prove that $\ominus$ produces valid changes on $A \to B$. Consider $df = f_2 \ominus f_1$ for $f_1, f_2 : A \to B$. For any valid input $da \triangleright a_1 \hookrightarrow a_2 : A$, we must show that df produces a valid output with the correct vertexes, that is, that $df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 : B$. Since $\oplus$ agrees with validity, a_2 equals $a_1 \oplus da$. By substituting away a_2 and df the thesis becomes $f_2\ (a_1 \oplus da) \ominus f_1\ a_1 \triangleright f_1\ a_1 \hookrightarrow f_2\ (a_1 \oplus da) : B$, which is true because $\ominus$ produces valid changes on B.

- $\mathbf{0}$ produces valid changes as proved in Lemma 13.1.8.

- We prove that change composition preserves validity on $A \to B$. That is, we must prove

  $$df_1\ a_1\ \mathbf{0}_{a_1} \circledcirc df_2\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_3\ a_2 : B$$

  for every $f_1, f_2, f_3, df_1, df_2, a_1, da, a_2$ satifsfying $df_1 \triangleright f_1 \hookrightarrow f_2 : A \to B$, $df_2 \triangleright f_2 \hookrightarrow f_3 : A \to B$ and $da \triangleright a_1 \hookrightarrow a_2 : A$.

  Because change composition preserves validity on B, it's enough to prove that (1) $df_1\ a_1\ \mathbf{0}_{a_1} \triangleright f_1\ a_1 \hookrightarrow f_2\ a_1 : B$ (2) $df_2\ a_1\ da \triangleright f_2\ a_1 \hookrightarrow f_3\ a_2 : B$. That is, intuitively, we create a composite change using $\circledcirc$, and it goes from $f_1\ a_1$ to $f_3\ a_2$ passing through $f_2\ a_1$. Part (1) follows because df_1 is a valid function change from f_1 to f_2, applied to a valid change $\mathbf{0}_{a_1}$ from a_1 to a_1. Part (2) follows because df_2 is a valid function change from f_2 to f_3, applied to a valid change da from a_1 to a_2. □

13.3.2 Change structures for products

We can define change structures on products $A \times B$, given change structures on A and B: a change on pairs is just a pair of changes; all other change structure definitions distribute componentwise the same way, and their correctness reduce to the correctness on components.

Change structures on n-ary products or records present no additional difficulty.

Definition 13.3.3 (Change structure for $A \times B$)
Given change structures $\widehat{A}$ and $\widehat{B}$ we define a change structure $\widehat{A} \times \widehat{B}$ on product $A \times B$.

(a) The change set is defined as: $\Delta(A \times B) = \Delta A \times \Delta B$.

(b) Validity is defined as

$$(da, db) \triangleright (a_1, b_1) \hookrightarrow (a_2, b_2) : A \times B = \\ (da \triangleright a_1 \hookrightarrow a_2 : A) \text{ and } (db \triangleright b_1 \hookrightarrow b_2 : B).$$

In other words, validity distributes componentwise: a product change is valid if each component is valid.

(c) We define change update by

$$(a_1, b_1) \oplus (da, db) = (a_1 \oplus da, b_1 \oplus db).$$

(d) We define difference by

$$(a_2, b_2) \ominus (a_1, b_1) = (a_2 \ominus a_1, b_2 \ominus b_1).$$

(e) We define $\mathbf{0}$ to distribute componentwise:

$$\mathbf{0}_{a,b} = (\mathbf{0}_a, \mathbf{0}_b).$$

(f) We define change composition to distribute componentwise:

$$(da_1, db_1) \circledcirc (da_2, db_2) = (da_1 \circledcirc da_2, db_1 \circledcirc db_2).$$ □

Lemma 13.3.4
Definition 13.3.3 defines a correct change structure $\widehat{A} \times \widehat{B}$. □

Proof. Since all these proofs are similar and spelling out their details does not make them clearer, we only give the first such proof in full.

- $\oplus$ agrees with validity on $A \times B$ because $\oplus$ agrees with validity on both A and B. For this property we give a full proof.

  For each $p_1, p_2 : A \times B$ and $dp \triangleright p_1 \hookrightarrow p_2 : A \times B$, we must show that $p_1 \oplus dp = p_2$. Instead of quantifying over pairs $p : A \times B$, we can quantify equivalently over components $a : A$, $b : B$. Hence, consider $a_1, a_2 : A$, $b_1, b_2 : B$, and changes da, db that are valid, that is, $da \triangleright a_1 \hookrightarrow a_2 : A$ and $db \triangleright b_1 \hookrightarrow b_2 : B$: We must show that

  $$(a_1, b_1) \oplus (da, db) = (a_2, b_2).$$

  That follows from $a_1 \oplus da = a_2$ (which follows from $da \triangleright a_1 \hookrightarrow a_2 : A$) and $b_1 \oplus db = b_2$ (which follows from $db \triangleright b_1 \hookrightarrow b_2 : B$).

- $\ominus$ produces valid changes on $A \times B$ because $\ominus$ produces valid changes on both A and B. We omit a full proof; the key step reduces the thesis

$$(a_2, b_2) \ominus (a_1, b_1) \triangleright (a_1, b_1) \hookrightarrow (a_2, b_2) : A \times B$$

  to $a_2 \ominus a_1 \triangleright a_1 \hookrightarrow a_2 : A$ and $b_2 \ominus b_1 \triangleright b_1 \hookrightarrow b_2 : B$ (where free variables range on suitable domains).

- $\mathbf{0}_{a,b}$ is correct, that is $(\mathbf{0}_a, \mathbf{0}_b) \triangleright (a, b) \hookrightarrow (a, b) : A \times B$, because $\mathbf{0}$ is correct on each component.

- Change composition is correct on $A \times B$, that is

$$(da_1 \circledcirc da_2, db_1 \circledcirc db_2) \triangleright (a_1, b_1) \hookrightarrow (a_3, b_3) : A \times B$$

  whenever $(da_1, db_1) \triangleright (a_1, b_1) \hookrightarrow (a_2, b_2) : A \times B$ and $(da_2, db_2) \triangleright (a_2, b_2) \hookrightarrow (a_3, b_3) : A \times B$, in essence because change composition is correct on both A and B. We omit details. □

13.4 Change structures for types and contexts

As promised, given change structures for base types we can provide change structures for all types:

Plugin Requirement 13.4.1 (Change structures for base types)
For each base type ι we must have a change structure $\widehat{\iota}$ defined on base set $[\![\iota]\!]$, based on the basic change structures defined earlier. □

Definition 13.4.2 (Change structure for types)
For each type τ we define a change structure $\widehat{\tau}$ on base set $[\![\tau]\!]$.

$$\widehat{\iota} = \dots$$
$$\widehat{\sigma \to \tau} = \widehat{\sigma} \to \widehat{\tau}$$

Lemma 13.4.3
Change sets and validity, as defined in Definition 13.4.2, give rise to the same basic change structures as the ones defined earlier in Definition 12.1.15, and to the change operations described in Fig. 13.1a. □

Proof. This can be verified by induction on types. For each case, it is sufficient to compare definitions. □

Lemma 13.4.4 ($\oplus$ agrees with validity)
If $dv \triangleright v_1 \hookrightarrow v_2 : \tau$ then $v_1 \oplus dv = v_2$. □

Proof. Because $\widehat{\tau}$ is a change structure and in change structures $\oplus$ agrees with validity. □

As shortly proved in Sec. 12.2, since $\oplus$ agrees with validity (Lemma 13.4.4) and $\mathcal{D}[\![-]\!]$ is correct (Theorem 12.2.2) we get Corollary 13.4.5:

Corollary 13.4.5 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
If $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![t]\!]\ \rho_1 \oplus [\![\mathcal{D}[\![t]\!]]\!]\ d\rho = [\![t]\!]\ \rho_2$. □

We can also define a change structure for environments. Recall that change structures for products define their operations to act on each component. Each change structure operation for environments acts "variable-wise". Recall that a typing context Γ is a list of variable assignment $x : \tau$. For each such entry, environments ρ and environment changes $d\rho$ contain a base entry $x = v$ where $v : [\![\tau]\!]$, and possibly a change $dx = dv$ where $dv : [\![\Delta\tau]\!]$.

Definition 13.4.6 (Change structure for environments)
To each context Γ we associate a change structure $\widehat{\Gamma}$, that extends the basic change structure from Definition 12.1.25. Operations are defined as shown in Fig. 13.1b. □

Base values v' in environment changes are redundant with base values v in base environments, because for valid changes $v = v'$. So when consuming an environment change, we choose arbitrarily to use v instead of v'. Alternatively, we could also use v' and get the same results for valid inputs. When producing an environment change, they are created to ensure validity of the resulting environment change.

Lemma 13.4.7
Definition 13.4.6 defines a correct change structure $\widehat{\Gamma}$ for each context Γ. □

Proof. All proofs are by structural induction on contexts. Most details are analogous to the ones for products and add no details, so we refer to our mechanization for most proofs.

However we show by induction that $\oplus$ agrees with validity. For the empty context there's a single environment $\varnothing : [\![\varepsilon]\!]$, so $\oplus$ returns the correct environment $\varnothing$. For the inductive case $\Gamma', x : \tau$, inversion on the validity judgment reduces our hypothesis to $dv \triangleright v_1 \hookrightarrow v_2 : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$, and our thesis to $(\rho_1, x = v_1) \oplus (d\rho, x = v_1, dx = dv) = (\rho_2, x = v_2)$, where v_1 appears both in the base environment and the environment change. The thesis follows because $\oplus$ agrees with validity on both Γ and τ. □

We summarize definitions on types in Fig. 13.1.

Finally, we can lift change operators from the semantic level to the syntactic level so that their meaning is coherent.

Definition 13.4.8 (Term-level change operators)
We define type-indexed families of change operators at the term level with the following signatures:

$$
\begin{aligned}
&\oplus_\tau : \tau \to \Delta\tau \to \tau \\
&\ominus_\tau : \tau \to \tau \to \Delta\tau \\
&\mathbf{0}_{\tau,-} : \tau \to \Delta\tau \\
&\circledcirc_\tau : \Delta\tau \to \Delta\tau \to \Delta\tau
\end{aligned}
$$

and definitions:

$$
\begin{aligned}
&tf_1 \ \oplus_{\sigma\to\tau} \ dtf = \lambda x \to tf_1\ x \oplus dtf\ x\ \mathbf{0}_x \\
&tf_2 \ \ominus_{\sigma\to\tau} \ tf_1 = \lambda x\ dx \to tf_2\ (x \oplus dx) \ominus tf_1\ x \\
&\mathbf{0}_{\sigma\to\tau,tf} = tf \ \ominus_{\sigma\to\tau} \ tf \\
&dtf_1 \ \circledcirc_{\sigma\to\tau} \ dtf_2 = \lambda x\ dx \to dtf_1\ x\ \mathbf{0}_x \circledcirc dtf_2\ x\ dx \\
&tf_1 \ \oplus_\iota \ dtf = \ldots \\
&tf_2 \ \ominus_\iota \ tf_1 = \ldots \\
&\mathbf{0}_{\iota,tf} = \ldots \\
&dtf_1 \ \circledcirc_\iota \ dtf_2 = \ldots
\end{aligned}
$$

Lemma 13.4.9 (Term-level change operators agree with change structures)
The following equations hold for all types τ, contexts Γ well-typed terms $\Gamma \vdash t_1, t_2 : \tau$, $\Delta\Gamma \vdash dt, dt_1, dt_2 : \Delta\tau$ and environments $\rho : [\![\Gamma]\!]$ $d\rho : [\![\Delta\Gamma]\!]$ such that all expressions are defined.

$$
\begin{aligned}
&[\![t_1 \ \oplus_\tau \ dt]\!] \ d\rho = [\![t_1]\!] \ d\rho \oplus [\![dt]\!] \ d\rho \\
&[\![t_2 \ \ominus_\tau \ t_1]\!] \ \rho = [\![t_2]\!] \ \rho \oplus [\![t_1]\!] \ \rho \\
&[\![\mathbf{0}_{\tau,t}]\!] \ \rho = \mathbf{0}_{[\![t]\!] \rho} \\
&[\![dt_1 \ \odot_\tau \ dt_2]\!] \ d\rho = [\![dt_1]\!] \ d\rho \odot [\![dt_2]\!] \ d\rho
\end{aligned}
$$

Proof. By induction on types and simplifying both sides of the equalities. The proofs for $\oplus$ and $\ominus$ must be done by simultaneous induction. □

At the time of writing, we have not mechanized the proof for $\odot$.

To define the lifting and prove it coherent on base types, we must add a further plugin requirement.

Plugin Requirement 13.4.10 (Term-level change operators for base types)
For each base type ι we define change operators as required by Definition 13.4.8 and satisfying requirements for Lemma 13.4.9. ▫

13.5 Development history

The proof presented in this and the previous chapter is an significant evolution of the original one by Cai et al. [2014]. While this formalization and the mechanization are both original with this thesis, some ideas were suggested by other (currently unpublished) developments by Yufei Cai and by Yann Régis-Gianas. Yufei Cai gives a simpler pen-and-paper set-theoretic proof by separating validity, while we noticed separating validity works equally well in a mechanized type theory and simplifies the mechanization. The first to use a two-sided validity relation was Yann Régis-Gianas, but using a big-step operational semantics, while we were collaborating on an ILC correctness proof for untyped λ-calculus (as in Appendix C). I gave the first complete and mechanized ILC correctness proof using two-sided validity, again for a simply-typed λ-calculus with a denotational semantics. Based on two-sided validity, I also reconstructed the rest of the theory of changes.

13.6 Chapter conclusion

In this chapter, we have seen how to define change operators, both on semantic values nad on terms, and what are their guarantees, via the notion of change structures. We have also defined change structures for groups, function spaces and products. Finally, we have explained and shown Corollary 13.4.5. We continue in next chapter by discussing how to reason syntactically about changes, before finally showing how to define some language plugins.

$$\boxed{\oplus_\tau : [\![\, \tau \to \Delta\tau \to \tau \,]\!]}$$
$$\boxed{\ominus_\tau : [\![\, \tau \to \tau \to \Delta\tau \,]\!]}$$
$$\boxed{\mathbf{0}_\_ : [\![\, \tau \to \Delta\tau \,]\!]}$$
$$\boxed{\circledcirc_\tau : [\![\, \Delta\tau \to \Delta\tau \to \Delta\tau \,]\!]}$$

$$\begin{array}{lll}
f_1 \ \oplus_{\sigma\to\tau} & df & = \lambda v \to f_1\ v \oplus df\ v\ \mathbf{0}_v \\
v_1 \ \oplus_\iota & dv & = \ldots \\
f_2 \ \ominus_{\sigma\to\tau} & f_1 & = \lambda v\ dv \to f_2\ (v \oplus dv) \ominus f_1\ v \\
v_2 \ \ominus_\iota & v_1 & = \ldots \\
\mathbf{0}_v & & = v\ \ominus_\tau\ v \\
dv_1 \ \circledcirc_\iota & dv_2 & = \ldots \\
df_1 \ \circledcirc_{\sigma\to\tau} & df_2 & = \lambda v\ dv \to df_1\ v\ \mathbf{0}_v \circledcirc df_2\ v\ dv
\end{array}$$

(a) Change structure operations on types (see Definition 13.4.2).

$$\boxed{\oplus_\Gamma : [\![\, \Gamma \to \Delta\Gamma \to \Gamma \,]\!]}$$
$$\boxed{\ominus_\Gamma : [\![\, \Gamma \to \Gamma \to \Delta\Gamma \,]\!]}$$
$$\boxed{\mathbf{0}_\_ : [\![\, \Gamma \to \Delta\Gamma \,]\!]}$$
$$\boxed{\circledcirc_\Gamma : [\![\, \Delta\Gamma \to \Delta\Gamma \to \Delta\Gamma \,]\!]}$$

$$\begin{array}{ll}
\varnothing \oplus \varnothing & = \varnothing \\
(\rho, x = v) \oplus (d\rho, x = v', dx = dv) & = (\rho \oplus d\rho, x = v \oplus dv) \\
\varnothing \ominus \varnothing & = \varnothing \\
(\rho_2, x = v_2) \ominus (\rho_1, x = v_1) & = (\rho_2 \ominus \rho_1, x = v_1, dx = v_2 \ominus v_1) \\
\mathbf{0}_\varnothing & = \varnothing \\
\mathbf{0}_{\rho, x=v} & = (\mathbf{0}_\rho, x = v, dx = \mathbf{0}_v)
\end{array}$$

$$\varnothing \circledcirc \varnothing = \varnothing$$
$$(d\rho_1, x = v_1, dx = dv_1) \circledcirc (d\rho_2, x = v_2, dx = dv_2) = (d\rho_1 \circledcirc d\rho_2, x = v_1, dx = dv_1 \circledcirc dv_2)$$

(b) Change structure operations on environments (see Definition 13.4.6).

Lemma 13.4.4 ($\oplus$ agrees with validity)
If $dv \triangleright v_1 \hookrightarrow v_2 : \tau$ then $v_1 \oplus dv = v_2$. □

Corollary 13.4.5 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
If $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![\, t \,]\!]\ \rho_1 \oplus [\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ d\rho = [\![\, t \,]\!]\ \rho_2$. □

Figure 13.1: Defining change structures.

Chapter 14

Equational reasoning on changes

In this chapter, we formalize equational reasoning directly on terms, rather than on semantic values (Sec. 14.1), and we discuss when two changes can be considered equivalent (Sec. 14.2). We also show, as an example, a simple change structure on lists and a derivative of *map* for it (Example 14.1.1).

To reason on terms, instead of describing the updated value of a term t by using an updated environment ρ_2, we substitute in t each variable x_i with expression $x_i \oplus dx_i$, to produce a term that computes the updated value of t, so that we can say that dx is a change from x to $x \oplus dx$, or that $df\ x\ dx$ is a change from $f\ x$ to $(f \oplus df)\ (x \oplus dx)$. Lots of the work in this chapter is needed to modify definitions, and go from using an updated environment to using substitution in this fashion.

To compare for equivalence terms that use changes, we can't use denotational equivalence, but must restrict to consider valid changes.

Comparing changes is trickier: most often we are not interested in whether two changes produce the same value, but whether two changes have the same source and destination. Hence, if two changes share source and destination we say they are *equivalent*. As we show in this chapter, operations that preserve validity also respect *change equivalence*, because for all those operations the source and destination of any output changes only depend on source and destination of input changes. Among the same source and destination there often are multiple changes, and the difference among them can affect how long a derivative takes, but not whether the result is correct.

We also show that change equivalence is a particular form of logical relation, a logical *partial equivalence relation* (PER). PERs are well-known to semanticists, and often used to study sets with invalid elements together with the appropriate equivalence on these sets.

The rest of the chapter expands on the details, even if they are not especially surprising.

14.1 Reasoning on changes syntactically

To define derivatives of primitives, we will often discuss validity of changes directly on programs, for instance saying that dx is a valid change from x to $x \oplus dx$, or that $f\ x \oplus df\ x\ dx$ is equivalent to $f\ (x \oplus dx)$ if all changes in scope are valid.

In this section we formalize these notions. We have not mechanized proofs involving substitutions, but we include them here, even though they are not especially interesting.

But first, we exemplify informally these notions.

Example 14.1.1 (Deriving *map* on lists)
Let's consider again the example from Sec. 11.3.3, in particular *dmap*. Recall that a list change *dxs* is valid for source *xs* if they have the same length and each element change is valid for its

corresponding element.

$$
\begin{aligned}
&map : (a \to b) \to List\ a \to List\ b \\
&map\ f\ Nil = Nil \\
&map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs) \\
&dmap : (a \to b) \to \Delta(a \to b) \to List\ a \to \Delta List\ a \to \Delta List\ b \\
&\quad \text{-- A valid list change has the same length as the base list:} \\
&dmap\ f\ df\ Nil\ Nil = Nil \\
&dmap\ f\ df\ (Cons\ x\ xs)\ (Cons\ dx\ dxs) = \\
&\quad Cons\ (df\ x\ dx)\ (dmap\ f\ df\ xs\ dxs) \\
&\quad \text{-- Remaining cases deal with invalid changes, and a dummy} \\
&\quad \text{-- result is sufficient.} \\
&dmap\ f\ df\ xs\ dxs = Nil
\end{aligned}
$$

In our example, one can show that *dmap* is a correct derivative for *map*. As a consequence, terms $map\ (f \oplus df)\ (xs \oplus dxs)$ and $map\ f\ xs \oplus dmap\ f\ df\ xs\ dxs$ are interchangeable in all valid contexts, that is, contexts that bind df and dxs to valid changes, respectively, for f and xs.

We sketch an informal proof directly on terms.

Proof sketch. We must show that $dy = dmap\ f\ df\ xs\ dxs$ is a change change from initial output $y_1 = map\ f\ xs$ to updated output $y_2 = map\ (f \oplus df)\ (xs \oplus dxs)$, for valid inputs df and dxs.

We proceed by structural induction on xs and dxs (technically, on a proof of validity of dxs). Since dxs is valid, those two lists have to be of the same length. If xs and dxs are both empty, $y_1 = dy = y_2 = Nil$ so dy is a valid change as required.

For the inductive step, both lists are *Cons* nodes, so we need to show that output change

$$dy = dmap\ f\ df\ (Cons\ x\ xs)\ (Cons\ dx\ dxs)$$

is a valid change from

$$y_1 = map\ f\ (Cons\ x\ xs)$$

to

$$y_2 = map\ (f \oplus df)\ (Cons\ (x \oplus dx)\ (xs \oplus dxs)).$$

To restate validity we name heads and tails of dy, y_1, y_2. If we write $dy = Cons\ dh\ dt$, $y_1 = Cons\ h_1\ t_1$ and $y_2 = Cons\ h_2\ t_2$, we need to show that dh is a change from h_1 to h_2 and dt is a change from t_1 to t_2.

Indeed, head change $dh = df\ x\ dx$ is a valid change from $h_1 = f\ x$ to $h_2 = f\ (x \oplus dx)$. And tail change $dt = dmap\ f\ df\ xs\ dxs$ is a valid change from $t_1 = map\ f\ xs$ to $t_2 = map\ (f \oplus df)\ (xs \oplus dxs)$ by induction. Hence dy is a valid change from y_1 to y_2. □

Hopefully this proof is already convincing, but it relies on undefined concepts. On a metalevel function, we could already make this proof formal, but not so on terms yet. In this section, we define the required concepts.

14.1.1 Denotational equivalence for valid changes

This example uses the notion of denotational equivalence for valid changes. We now proceed to formalize it. For reference, we recall denotational equivalence of terms, and then introduce its restriction:

Definition A.2.5 (Denotational equivalence)
We say that two terms $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$ are denotationally equal, and write $\Gamma \vDash t_1 \cong t_2 : \tau$ (or sometimes $t_1 \cong t_2$), if for all environments $\rho : [\![\Gamma]\!]$ we have that $[\![t_1]\!]\ \rho = [\![t_2]\!]\ \rho$. □

For open terms t_1, t_2 that depend on changes, denotational equivalence is too restrictive, since it requires t_1 and t_2 to also be equal when the changes they depend on are not valid. By restricting denotational equivalence to valid environment changes, and terms to depend on contexts, we obtain the following definition.

Definition 14.1.2 (Denotational equivalence for valid changes)
For any context Γ and type τ, we say that two terms $\Delta\Gamma \vdash t_1 : \tau$ and $\Delta\Gamma \vdash t_2 : \tau$ are *denotationally equal for valid changes* and write $\Delta\Gamma \vDash t_1 \cong_\Delta t_2 : \tau$ if, for all valid environment changes $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ we have that t_1 and t_2 evaluate in environment $d\rho$ to the same value, that is, $[\![t_1]\!]\ d\rho = [\![t_2]\!]\ d\rho$. □

Example 14.1.3
Terms $f\ x \oplus df\ x\ dx$ and $f\ (x \oplus dx)$ are denotationally equal for valid changes (for any types σ, τ): $\Delta(f : \sigma \to \tau, x : \sigma) \vDash f\ x \oplus df\ x\ dx \cong_\Delta f\ (x \oplus dx) : \tau$. □

Example 14.1.4
One of our claims in Example 14.1.1 can now be written as

$$\Delta\Gamma \vDash map\ (f \oplus df)\ (xs \oplus dxs) \cong_\Delta map\ f\ xs \oplus dmap\ f\ df\ xs\ dxs : List\ b$$

for a suitable context $\Gamma = f : List\ \sigma \to List\ \tau, xs : List\ \sigma, map : (\sigma \to \tau) \to List\ \sigma \to List\ \tau$ (and for any types σ, τ). □

Arguably, the need for a special equivalence is a defect in our semantics of change programs; it might be more preferable to make the type of changes abstract throughout the program (except for derivatives of primitives, which must inspect derivatives), but this is not immediate, especially in a module system like Haskell. Other possible alternatives are discussed in Sec. 15.4.

14.1.2 Syntactic validity

Next, we define *syntactic validity*, that is, when a change *term* dt is a (valid) change from source term t_1 to destination t_2. Intuitively, dt is valid from t_1 to t_2 if dt, t_1 and t_2, evaluated all against the same valid environment change $d\rho$, produce a valid change, its source and destination. Formally:

Definition 14.1.5 (Syntactic validity)
We say that term $\Delta\Gamma \vdash dt : \Delta\tau$ is a (syntactic) change from $\Delta\Gamma \vdash t_1 : \tau$ to $\Delta\Gamma \vdash t_2 : \tau$, and write $\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 : \tau$, if

$$\forall d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma.\ [\![dt]\!]\ d\rho \triangleright [\![t_1]\!]\ d\rho \hookrightarrow [\![t_2]\!]\ d\rho : \tau.$$ □

Notation 14.1.6
We often simply say that dt is a change from t_1 to t_2, leaving everything else implicit when not important. □

Using syntactic validity, we can show for instance that dx is a change from x to $x \oplus dx$, that $df\ x\ dx$ is a change from $f\ x$ to $(f \oplus df)\ (x \oplus dx)$; both examples follow from a general statement about $\mathcal{D}[\![-]\!]$ (Theorem 14.1.9). Our earlier informal proof of the correctness of *dmap* (Example 14.1.1) can also be justified in terms of syntactic validity.

Just like (semantic) $\oplus$ agrees with validity, term-level (or syntactic) $\oplus$ agrees with syntactic validity, up to denotational equivalence for valid changes.

Lemma 14.1.7 (Term-level ⊕ agrees with syntactic validity)
If dt is a change from t_1 to t_2 ($\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 : \tau$) then $t_1 \oplus dt$ and t_2 are denotationally equal for valid changes ($\Delta\Gamma \vDash t_1 \oplus dt \cong_\Delta t_2 : \tau$). □

Proof. Follows because term-level ⊕ agrees with semantic ⊕ (Lemma 13.4.9) and ⊕ agrees with validity. In more detail: fix an arbitrary valid environment change $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$. First, we have $[\![dt]\!]\ d\rho \triangleright [\![t_1]\!]\ d\rho \hookrightarrow [\![t_2]\!]\ d\rho : \tau$ because of syntactic validity. Then we conclude with this calculation:

$$
\begin{aligned}
& [\![t_1 \oplus dt]\!]\ d\rho \\
= \quad & \{ \text{term-level } \oplus \text{ agrees with } \oplus \text{ (Lemma 13.4.9)} \} \\
& [\![t_1]\!]\ d\rho \oplus [\![dt]\!]\ d\rho \\
= \quad & \{ \oplus \text{ agrees with validity} \} \\
& [\![t_2]\!]\ d\rho
\end{aligned}
$$

□

Beware that the definition of $\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 : \tau$ evaluates all terms against change environment $d\rho$, containing separately base values and changes. In contrast, if we use validity in the change structure for $[\![\Gamma]\!] \to [\![\tau]\!]$, we evaluate different terms against different environments. That is why we have that dx is a change from x to $x \oplus dx$ (where $x \oplus dx$ is evaluated in environment $d\rho$), while $[\![dx]\!]$ is a valid change from $[\![x]\!]$ to $[\![x]\!]$ (where the destination $[\![x]\!]$ gets applied to environment ρ_2).

Is syntactic validity trivial? Without needing syntactic validity, based on earlier chapters one can show that dv is a valid change from v to $v \oplus dv$, or that $df\ v\ dv$ is a valid change from $f\ v$ to $(f \oplus df)\ (v \oplus dv)$, or further examples. But that's all about values. In this section we are just translating these notions to the level of terms, and formalize them.

Our semantics is arguably (intuitively) a trivial one, similar to a metainterpreter interpreting object-language functions in terms of metalanguage functions: our semantics simply embeds an object-level λ-calculus into a meta-level and more expressive λ-calculus, mapping for instance $\lambda f\ x \to f\ x$ (an AST) into $\lambda f\ v \to f\ v$ (syntax for a metalevel function). Hence, proofs in this section about syntactic validity deal mostly with this translation. We don't expect the proofs to give special insights, and we expect most development would keep such issues informal (which is certainly reasonable).

Nevertheless, we write out the statements to help readers refer to them, and write out (mostly) full proofs to help ourselves (and readers) verify them. Proofs are mostly standard but with a few twists, since we must often consider and relate *three* computations: the computation on initial values and the ones on the change and on updated values.

We're also paying a proof debt. Had we used substitution and small step semantics, we'd have directly simple statements on terms, instead of trickier ones involving terms and environments. We produce those statements now.

Differentiation and syntactic validity

We can also show that $\mathcal{D}[\![t]\!]$ produces a syntactically valid change from t, but we need to specify its destination. In general, $\mathcal{D}[\![t]\!]$ is not a change from t to t. The destination must evaluate to the updated value of t; to produce a term that evaluates to the right value, we use substitution. If the only free variable in t is x, then $\mathcal{D}[\![t]\!]$ is a syntactic change from t to $t\,[x := x \oplus dx]$. To repeat the same for all variables in context Γ, we use the following notation.

Notation 14.1.8
We write $t\,[\Gamma := \Gamma \oplus \Delta\Gamma]$ to mean $t\,[x_1 := x_1 \oplus dx_1, x_2 := x_2 \oplus dx_2, \ldots, x_n := x_n \oplus dx_n]$. □

Theorem 14.1.9 ($\mathcal{D}[\![-]\!]$ is correct, syntactically)
For any well-typed term $\Gamma \vdash t : \tau$, term $\mathcal{D}[\![t]\!]$ is a syntactic change from t to $t\,[\Gamma := \Gamma \oplus \Delta\Gamma]$. □

We present the following straightforward (if tedious) proof (formal but not mechanized).

Proof. Let $t_2 = t\,[\Gamma := \Gamma \oplus \Delta\Gamma]$. Take any $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$. We must show that $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ d\rho \hookrightarrow [\![t_2]\!]\ d\rho : \tau$.

Because $d\rho$ extend ρ_1 and t only needs entries in ρ_1, we can show that $[\![t]\!]\ d\rho = [\![t]\!]\ \rho_1$, so our thesis becomes $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t_2]\!]\ d\rho : \tau$.

Because $\mathcal{D}[\![-]\!]$ is correct (Theorem 12.2.2) we know that $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \tau$; that's almost our thesis, so we must only show that $[\![t_2]\!]\ d\rho = [\![t]\!]\ \rho_2$. Since $\oplus$ agrees with validity and $d\rho$ is valid, we have that $\rho_2 = \rho_1 \oplus d\rho$, so our thesis is now the following equation, which we leave to Lemma 14.1.10:

$$[\![t]\!]\ [\Gamma := \Gamma \oplus \Delta\Gamma]\ d\rho = [\![t]\!]\ (\rho_1 \oplus d\rho). \qquad \text{fl}$$

Here's the technical lemma to complete the proof.

Lemma 14.1.10
For any $\Gamma \vdash t : \tau$, and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ we have

$$[\![t]\!]\ [\Gamma := \Gamma \oplus \Delta\Gamma]\ d\rho = [\![t]\!]\ (\rho_1 \oplus d\rho).$$ □

Proof. This follows from the structure of valid environment changes, because term-level $\oplus$ (used on the left-hand side) agrees with value-level $\oplus$ (used on the right-hand side) by Lemma 13.4.9, and because of the substitution lemma.

More formally, we can show the thesis by induction over environments: for empty environments, the equations reduces to $[\![t]\!]\ \varnothing = [\![t]\!]\ \varnothing$. For the case of $\Gamma, x : \sigma$ (where $x \notin \Gamma$), the thesis can be rewritten as

$$[\![(t\,[\Gamma := \Gamma \oplus \Delta\Gamma])\,[x := x \oplus dx]]\!]\ (d\rho, x = v_1, dx = dv) = [\![t]\!]\ (\rho_1 \oplus d\rho, x = v_1 \oplus dv).$$

We prove it via the following calculation.

$$\begin{aligned}
& [\![(t\,[\Gamma := \Gamma \oplus \Delta\Gamma])\,[x := x \oplus dx]]\!]\ (d\rho, x = v_1, dx = dv) \\
= \quad & \{ \text{Substitution lemma on } x. \} \\
& [\![(t\,[\Gamma := \Gamma \oplus \Delta\Gamma])]\!]\ (d\rho, x = ([\![x \oplus dx]\!]\ (d\rho, x = v_1, dx = dv)), dx = dv) \\
= \quad & \{ \text{Term-level } \oplus \text{ agrees with } \oplus \text{ (Lemma 13.4.9). } \} \\
& \quad \{ \text{Then simplify } [\![x]\!] \text{ and } [\![dx]\!]. \} \\
& [\![(t\,[\Gamma := \Gamma \oplus \Delta\Gamma])]\!]\ (d\rho, x = v_1 \oplus dv, dx = dv) \\
= \quad & \{ t\,[\Gamma := \Gamma \oplus \Delta\Gamma] \text{ does not mention } dx. \} \\
& \quad \{ \text{So we can modify the environment entry for } dx. \} \\
& \quad \{ \text{This makes the environment into a valid environment change. } \} \\
& [\![(t\,[\Gamma := \Gamma \oplus \Delta\Gamma])]\!]\ (d\rho, x = v_1 \oplus dv, dx = \mathbf{0}_{v_1 \oplus dv}) \\
= \quad & \{ \text{By applying the induction hypothesis and simplifying } \mathbf{0} \text{ away. } \} \\
& [\![t]\!]\ (\rho_1 \oplus d\rho, x = v_1 \oplus dv)
\end{aligned}$$

□

14.2 Change equivalence

To optimize programs manipulate changes, we often want to replace a change-producing term by another one, while preserving the overall program meaning. Hence, we define an equivalence on valid changes that is preserved by change operations, that is (in spirit) a *congruence*. We call this relation *(change) equivalence*, and refrain from using other equivalences on changes.

Earlier (say, in Sec. 15.3) we have sometimes required that changes be equal, but that's often too restrictive.

Change equivalence is defined in terms of validity, to ensure that validity-preserving operations preserve change equivalence: If two changes dv_1 and dv_2 are equivalent, one can be substituted for the other in a validity-preserving context. We can define this once for all change structures, hence in particular for values and environments.

Definition 14.2.1 (Change equivalence)
Given a change structure $\widehat{V}$, changes $dv_a, dv_b : \Delta V$ are equivalent relative to source $v_1 : V$ (written $dv_a =_\Delta dv_b \triangleright v_1 \hookrightarrow v_2 : V$) if and only if there exists v_2 such that both dv_a and dv_b are valid from v_1 to v_2 (that is $dv_a \triangleright v_1 \hookrightarrow v_2 : V$, $dv_b \triangleright v_1 \hookrightarrow v_2 : V$). □

Notation 14.2.2
When not ambiguous we often abbreviate $dv_a =_\Delta dv_b \triangleright v_1 \hookrightarrow v_2 : V$ as $dv_a =_\Delta^{v_1} dv_b$ or $dv_a =_\Delta dv_b$.

Two changes are often equivalent relative to a source but not others. Hence $dv_a =_\Delta dv_b$ is always an abbreviation for change equivalence for a specific source. □

Example 14.2.3
For instance, we later use a change structure for integers using both replacement changes and differences (Example 12.1.6). In this structure, change 0 is nil for all numbers, while change !5 ("bang 5") replaces any number with 5. Hence, changes 0 and !5 are equivalent only relative to source 5, and we write $0 =_\Delta^5 !5$. □

By applying definitions, one can verify that change equivalence relative to a source v is a symmetric and transitive relation on ΔV. However, it is not an equivalence relation on ΔV, because it is only reflexive on changes valid for source v. Using the set-theoretic concept of subset we can then state the following lemma (whose proof we omit as it is brief):

Lemma 14.2.4 ($=_\Delta$ is an equivalence on valid changes)
For each set V and source $v \in V$, change equivalence relative to source v is an equivalence relation over the set of changes

$$\{dv \in \Delta V \mid dv \text{ is valid with source } v\}.$$

□

We elaborate on this peculiar sort of equivalence in Sec. 14.2.3.

14.2.1 Preserving change equivalence

Change equivalence relative to a source v is respected, in an appropriate sense, by all validity-preserving expression contexts that accept changes with source v. To explain what this means we study an example lemma: we show that because valid function changes preserve validity, they also respect change equivalence. At first, we use "(expression) context" informally to refer to expression contexts in the metalanguage. Later, we'll extend our discussion to actual expression contexts in the object language.

Lemma 14.2.5 (Valid function changes respect change equivalence)
Any valid function change

$$df \triangleright f_1 \hookrightarrow f_2 : A \to B$$

respects change equivalence: if $dv_a =_\Delta dv_b \triangleright v_1 \hookrightarrow v_2 : A$ then $df\ v_1\ dv_a =_\Delta df\ v_1\ dv_b \triangleright f_1\ v_1 \hookrightarrow f_2\ v_2 : B$. We also say that (expression) *context* $df\ v_1$ – respects change equivalence. □

Proof. The thesis means that $df\ v_1\ dv_a \triangleright f_1\ v_1 \hookrightarrow f_2\ v_2 : B$ and $df\ v_1\ dv_b \triangleright f_1\ v_1 \hookrightarrow f_2\ v_2 : B$. Both equivalences follow in one step from validity of df, dv_a and dv_b. □

This lemma holds because the source and destination of $df\ v_1\ dv$ don't depend on dv, only on its source and destination. Source and destination are shared by equivalent changes. Hence, validity-preserving functions map equivalent changes to equivalent changes.

In general, all operations that preserve validity also respect *change equivalence*, because for all those operations, the source and destination of any output changes, and the resulting value, only depend on source and destination of input changes.

However, Lemma 14.2.5 does *not* mean that $df\ v_1\ dv_a = df\ v_1\ dv_b$, because there can be multiple changes with the same source and destination. For instance, say that dv_a is a list change that removes an element and readds it, and dv_b is a list change that describes no modification. They are both nil changes, but a function change might handle them differently.

Moreover, we only proved that context $df\ v_1$ – respects change equivalence relative to source v_1. If value v_3 differs from v_1, $df\ v_3\ dv_a$ and $df\ v_3\ dv_b$ need not be equivalent. Hence, we say that context $df\ v_1$ *accepts changes* with source v_1. More in general, a context accepts changes with source v_1 if it preserves validity for changes with source v_1; we can say informally that all such contexts also respect change equivalence.

Another example: context $v_1 \oplus -$ also accepts changes with source v_1. Since this context produces a base value and not a change, it maps equivalent changes to equal results:

Lemma 14.2.6 ($\oplus$ respects change equivalence)
If $dv_a =_\Delta dv_b \triangleright v_1 \hookrightarrow v_2 : V$ then $v_1 \oplus -$ respects the equivalence between dv_a and dv_b, that is, $v_1 \oplus dv_a = v_1 \oplus dv_b$. □

Proof. $v_1 \oplus dv_a = v_2 = v_1 \oplus dv_b$. □

There are more contexts that preserve equivalence. As discussed, function changes preserve contexts, and $\mathcal{D}[\![-]\!]$ produces functions changes, so $\mathcal{D}[\![t]\!]$ preserves equivalence on its environment, and on any of its free variables.

Lemma 14.2.7 ($\mathcal{D}[\![-]\!]$ preserves change equivalence)
For any term $\Gamma \vdash t : \tau$, $\mathcal{D}[\![t]\!]$ preserves change equivalence of environments: for all $d\rho_a =_\Delta d\rho_b \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ we have $[\![\mathcal{D}[\![t]\!]]\!]\ d\rho_a =_\Delta [\![\mathcal{D}[\![t]\!]]\!]\ d\rho_b \triangleright [\![t]\!]\ \rho_1 \hookrightarrow [\![t]\!]\ \rho_2 : \Gamma \to \tau$. □

Proof. To verify this, just apply correctness of differentiation to both changes $d\rho_a$ and $d\rho_b$. □

To show more formally in what sense change equivalence is a congruence, we first lift change equivalence to terms (Definition 14.2.9), similarly to syntactic change validity in Sec. 14.1.2. To do so, we first need a notation for *one-sided* or *source-only* validity:

Notation 14.2.8 (One-sided validity)
We write $dv \triangleright v_1 : V$ to mean there exists v_2 such that $dv \triangleright v_1 \hookrightarrow v_2 : V$. We will reuse existing conventions and write $dv \triangleright v_1 : \tau$ instead of $dv \triangleright v_1 : [\![\tau]\!]$ and $d\rho \triangleright \rho_1 : \Gamma$ instead of $d\rho \triangleright \rho_1 : [\![\Gamma]\!]$. □

Definition 14.2.9 (Syntactic change equivalence)
Two change terms $\Delta\Gamma \vdash dt_a : \Delta\tau$ and $\Delta\Gamma \vdash dt_b : \Delta\tau$ are change equivalent, relative to source $\Gamma \vdash t : \tau$, if for all valid environment changes $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ we have that

$$[\![dt_a]\!]\ d\rho =_\Delta [\![dt_b]\!]\ d\rho \triangleright [\![t]\!]\ \rho_1 : \tau.$$

We write then $\Gamma \models dt_a =_\Delta dt_b \blacktriangleright t : \tau$ or $dt_a =^t_\Delta dt_b$, or simply $dt_a =_\Delta dt_b$. □

Saying that dt_a and dt_b are equivalent relative to t does not specify the destination of dt_a and dt_b, only their source. The only reason is to simplify the statement and proof of Theorem 14.2.10.

If two change terms are change equivalent with respect to the right source, we can replace one for the other in an expression context to optimize a program, as long as the expression context is validity-preserving and accepts the change.

In particular, substituting into $\mathcal{D}[\![\, t \,]\!]$ preserves syntactic change equivalence, according to the following theorem (for which we have only a pen-and-paper formal proof).

Theorem 14.2.10 ($\mathcal{D}[\![-]\!]$ preserves syntactic change equivalence)
For any equivalent changes $\Gamma \models s \blacktriangleright ds_a =_\Delta ds_b : \sigma$, and for any term t typed as $\Gamma, x : \sigma \vdash t : \tau$, we can produce equivalent results by substituting into $\mathcal{D}[\![\, t \,]\!]$ either s and ds_a or s and ds_b:

$$\Gamma \models \mathcal{D}[\![\, t \,]\!]\ [x := s, dx := ds_a] =_\Delta \mathcal{D}[\![\, t \,]\!]\ [x := s, dx := ds_b] \blacktriangleright t\,[x := s] : \tau. \qquad \square$$

Proof sketch. The thesis holds because $\mathcal{D}[\![-]\!]$ preserves change equivalence Lemma 14.2.7. A formal proof follows through routine (and tedious) manipulations of bindings. In essence, we can extend a change environment $d\rho$ for context Γ to equivalent environment changes for context $\Gamma, x : \sigma$ with the values of ds_a and ds_b. The tedious calculations follow. $\square$

Proof. Assume $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$. Because ds_a and ds_b are change-equivalent we have

$$[\![\, ds_a \,]\!]\ d\rho =_\Delta [\![\, ds_b \,]\!]\ d\rho \triangleright [\![\, s \,]\!]\ \rho_1 : \sigma.$$

Moreover, $[\![\, s \,]\!]\ \rho_1 = [\![\, s \,]\!]\ d\rho$ because $d\rho$ extends ρ_1. We'll use this equality without explicit mention.

Hence, we can construct change-equivalent environments for evaluating $\mathcal{D}[\![\, t \,]\!]$, by combining $d\rho$ and the values of respectively ds_a and ds_b:

$$\begin{aligned}(d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_a \,]\!]\ d\rho) =_\Delta \\ (d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_b \,]\!]\ d\rho) \\ \triangleright (\rho_1, x = [\![\, s \,]\!]\ \rho_1) : (\Gamma, x : \sigma). \end{aligned} \tag{14.1}$$

This environment change equivalence is respected by $\mathcal{D}[\![\, t \,]\!]$, hence:

$$\begin{aligned}[\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_a \,]\!]\ d\rho) =_\Delta \\ [\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_b \,]\!]\ d\rho) \\ \triangleright [\![\, t \,]\!]\ (\rho_1, x = [\![\, s \,]\!]\ \rho_1) : \Gamma \to \tau. \end{aligned} \tag{14.2}$$

We want to deduce the thesis by applying to this statement the substitution lemma for denotational semantics: $[\![\, t \,]\!]\ (\rho, x = [\![\, s \,]\!]\ \rho) = [\![\, t\,[x := s] \,]\!]\ \rho$.

To apply the substitution lemma to the substitution of dx, we must adjust Eq. (14.2) using soundness of weakening. We get:

$$\begin{aligned}[\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_a \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho)) =_\Delta \\ [\![\, \mathcal{D}[\![\, t \,]\!] \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho, dx = [\![\, ds_b \,]\!]\ (d\rho, x = [\![\, s \,]\!]\ d\rho)) \\ \triangleright [\![\, t \,]\!]\ (\rho_1, x = [\![\, s \,]\!]\ \rho_1) : \Gamma \to \tau. \end{aligned} \tag{14.3}$$

This equation can now be rewritten (by applying the substitution lemma to the substitutions of dx and x) to the following one:

$$[\![(\mathcal{D}[\![t]\!]\ [dx := ds_a]\ [x := s])]\!]\ d\rho =_\Delta [\![(\mathcal{D}[\![t]\!]\ [dx := ds_b]\ [x := s])]\!]\ d\rho \triangleright [\![t\ [x := s]]\!]\ \rho_1 : \Gamma \to \tau. \quad (14.4)$$

Since x is not in scope in s, ds_a, ds_b, we can permute substitutions to conclude that:

$$\Gamma \models \mathcal{D}[\![t]\!]\ [x := s, dx := ds_a] =_\Delta \mathcal{D}[\![t]\!]\ [x := s, dx := ds_b] \blacktriangleright t\ [x := s] : \tau$$

as required. □

In this theorem, if x appears once in t, then dx appears once in $\mathcal{D}[\![t]\!]$ (this follows by induction on t), hence $\mathcal{D}[\![t]\!]\ [x := s, dx := -]$ produces a one-hole expression context.

Further validity-preserving contexts There are further operations that preserve validity. To represent terms with "holes" where other terms can be inserted, we can define *one-level contexts* F, and contexts E, as is commonly done:

$$F ::= [\,]\ t\ dt \mid ds\ t\ [\,] \mid \lambda x\ dx \to [\,] \mid t \oplus [\,] \mid dt_1 \circledcirc [\,] \mid [\,] \circledcirc dt_2$$
$$E ::= [\,] \mid F\ [E]$$

If $dt_1 =_\Delta dt_2 \triangleright t_1 \hookrightarrow t_2 : \tau$ and our context E accepts changes from t_1, then $F\ [dt_1]$ and $F\ [dt_2]$ are change equivalent. It is easy to prove such a lemma for each possible shape of one-level context F, both on values (like Lemmas 14.2.5 and 14.2.6) and on terms. We have been unable to state a more general theorem because it's not clear how to formalize the notion of a context accepting a change in general: the syntax of a context does not always hint at the validity proofs embedded.

Cai et al. [2014] solve this problem for metalevel contexts by typing them with dependent types, but as discussed the overall proof is more awkward. Alternatively, it appears that the use of dependent types in Chapter 18 also ensures that change equivalence is a congruence (though at present this is still a conjecture), without overly complicating correctness proofs. However, it is not clear whether such a type system can be expressive enough without requiring additional coercions. Consider a change dv_1 from v_1 to $v_1 \oplus dv_1$, a value v_2 which is known to be (propositionally) equal to $v_1 \oplus dv_1$, and a change dv_2 from v_2 to v_3. Then, term $dv_1 \circledcirc dv_2$ is not type correct (for instance in Agda): the typechecker will complain that dv_1 has destination $v_1 \oplus dv_1$ while dv_2 has source v_2. When working in Agda, to solve this problem we can explicitly coerce terms through propositional equalities, and can use Agda to prove such equalities in the first place. We leave the design of a sufficiently expressive object language where change equivalence is a congruence for future work.

14.2.2 Sketching an alternative syntax

If we exclude composition, we can sketch an alternative syntax which helps construct a congruence on changes. The idea is to manipulate, instead of changes alone, pairs of sources $v \in V$ and valid changes $\{dv \mid dv \triangleright v : V\}$.

$$t ::= \mathbf{src}\ dt \mid \mathbf{dst}\ dt \mid x \mid t\ t \mid \lambda x \to t$$
$$dt ::= dt\ dt \mid \mathbf{src}\ dt \mid \lambda dx \to dt \mid dx$$

Adapting differentiation to this syntax is easy:

$$\mathcal{D}[\![x]\!] = dx$$
$$\mathcal{D}[\![s\ t]\!] = \mathcal{D}[\![s]\!]\ \mathcal{D}[\![t]\!]$$
$$\mathcal{D}[\![\lambda x \to t]\!] = \mathcal{D}[\![\lambda dx \to dt]\!]$$

Derivatives of primitives only need to use $dst\ dt$ instead of $t \oplus dt$ and $src\ dt$ instead of t whenever dt is a change for t.

With this syntax, we can define change expression contexts, which can be filled in by change expressions:

$$E ::= src\ dE \mid dst\ dE \mid E\ t \mid t\ E \mid \lambda x \to E$$
$$dE ::= dt\ dE \mid dE\ dt \mid \lambda dx \to dE \mid [\,]$$

We conjecture change equivalence is a congruence with respect to contexts dE, and that contexts E map change-equivalent changes to results that are denotationally equivalent for valid changes. We leave a proof to future work, but we expect it to be straightforward. It also appears straightforward to provide an isomorphism between this syntax and the standard one.

However, extending such contexts with composition does not appear trivial: contexts such as $dt \circledcirc dE$ or $dE \circledcirc dt$ only respect validity when the changes sources and destinations align correctly.

We make no further use of this alternative syntax in this work.

14.2.3 Change equivalence is a PER

Readers with relevant experience will recognize that change equivalence is a partial equivalence relation (PER) [Mitchell, 1996, Ch. 5]. It is standard to use PERs to identify valid elements in a model [Harper, 1992]. In this section, we state the connection, showing that change equivalence is not an ad-hoc construction, so that mathematical constructions using PERs can be adapted to use change equivalence.

We recall the definition of a PER:

Definition 14.2.11 (Partial equivalence relation (PER))
A relation $R \subseteq S \times S$ is a partial equivalence relation if it is symmetric (if aRb then bRa) and transitive (if aRb and bRc then aRc). □

Elements related to another are also related to themselves: If aRb then aRa (by transitivity: aRb, bRa, hence aRa). So a PER on S identifies a subset of valid elements of S. Since PERs are equivalence relations on that subset, they also induce a (partial) partition of elements of S into equivalence classes of change-equivalent elements.

Lemma 14.2.12 ($=_\Delta$ is a PER)
Change equivalence relative to a source $a : A$ is a PER on set ΔA. □

Proof. A restatement of Lemma 14.2.4. □

Typically, one studies *logical PERs*, which are logical relations and PERs at the same time [Mitchell, 1996, Ch. 8]. In particular, with a logical PER two functions are related if they map related inputs to related outputs. This helps showing that a PERs is a congruence. Luckily, our PER is equivalent to such a definition.

Lemma 14.2.13 (Alternative definition for $=_\Delta$)
Change equivalence is equivalent to the following logical relation:

$$dv_a =_\Delta dv_b \rhd v_1 \hookrightarrow v_2 : \iota \qquad =_{def}$$
$$dv_a \rhd v_1 \hookrightarrow v_2 : \iota \text{ and } dv_a \rhd v_1 \hookrightarrow v_2 : \iota$$

$$
\begin{aligned}
&df_a =_\Delta df_b \triangleright f_1 \hookrightarrow f_2 : \sigma \to \tau =_{def} \\
&\quad \forall dv_a =_\Delta dv_b \triangleright v_1 \hookrightarrow v_2 : \sigma. \\
&\quad df_a\ v_1\ dv_a =_\Delta df_b\ v_2\ dv_b \triangleright f_1\ v_1 \hookrightarrow f_2\ v_2 : \tau
\end{aligned}
$$

Proof. By induction on types. □

14.3 Chapter conclusion

In this chapter, we have put on a more solid foundation forms of reasoning about changes on terms, and defined an appropriate equivalence on changes.

Chapter 15

Extensions and theoretical discussion

In this chapter we collect discussion of a few additional topics related to ILC that do not suffice for standalone chapters. We show how to differentiation general recursion Sec. 15.1, we exhibit a function change that is not valid for any function (Sec. 15.2), we contrast our representation of function changes with *pointwise* function changes (Sec. 15.3), and we compare our formalization with the one presented in [Cai et al., 2014] (Sec. 15.4).

15.1 General recursion

This section discusses informally how to differentiate terms using general recursion and what is the behavior of the resulting terms.

15.1.1 Differentiating general recursion

Earlier we gave a rule for differentiating (non-recursive) **let**:

$$\begin{aligned}\mathcal{D}[\![\,\textbf{let } x = t_1 \textbf{ in } t_2\,]\!] = \textbf{let } x &= t_1\\ dx &= \mathcal{D}[\![\,t_1\,]\!]\\ \textbf{in } &\ \mathcal{D}[\![\,t_2\,]\!]\end{aligned}$$

It turns out that we can use the same rule also for recursive **let**-bindings, which we write here (and only here) **letrec** for emphasis:

$$\begin{aligned}\mathcal{D}[\![\,\textbf{letrec } x = t_1 \textbf{ in } t_2\,]\!] = \textbf{letrec } x &= t_1\\ dx &= \mathcal{D}[\![\,t_1\,]\!]\\ \textbf{in } &\ \mathcal{D}[\![\,t_2\,]\!]\end{aligned}$$

This rule applies also to recursive top-level definitions, since in our scenario they can be understood as uses of **letrec**.

Example 15.1.1
In Example 14.1.1 we presented a derivative *dmap* for *map*; since we wrote *dmap* by hand, we had to prove that *dmap* is a derivative for *map*.

We can instead obtain *dmap* by deriving *map* with our new rule for recursive functions:[1]

```
dmap f df Nil Nil = Nil
dmap f df (Cons x xs) (Cons dx dxs) =
  Cons (df x dx) (dmap f df xs dxs)
  -- Other cases deal with invalid changes.
dmap f df xs dxs = Nil
```

However, derivative *dmap* is not asymptotically faster than *map*, and this is typical: Derivatives of recursive functions produced using this rule are often not asymptotically faster, even when we consider less trivial change structures. Deriving **letrec** $x = t_1$ **in** t_2 can still be useful if $\mathcal{D}[\![t_1]\!]$ and/or $\mathcal{D}[\![t_2]\!]$ is faster than its base term, but during our work we focus mostly on using structural recursion. Alternatively, in Chapter 11 and Sec. 11.2 we have shown how to incrementalize functions (including recursive ones) using equational reasoning.

In general, when we invoke *dmap* on a change *dxs* from xs_1 to xs_2, it is important that xs_1 and xs_2 are similar enough that enough computation can be reused. Say that xs_1 = *Cons* 2 (*Cons* 3 (*Cons* 4 *Nil*)) and xs_2 = *Cons* 1 (*Cons* 2 (*Cons* 3 (*Cons* 4 *Nil*))): in this case, a change modifying each element of xs_1, and then replacing *Nil* by *Cons* 4 *Nil*, would be inefficient to process, and naive incrementalization would produce this scenario. In this case, it is clear that a preferable change should simply insert 1 at the beginning of the list, as illustrated in Sec. 11.3.2 (though we have omitted the straightforward definition of *dmap* for such a change structure). In approaches like self-adjusting computation, this is ensured by using memoization. In our approach, instead, we rely on changes that are nil or small to detect when a derivative can reuse input computation.

The same problem affects naive attempts to incrementalize, for instance, a simple factorial function; we omit details. Because of these issues, we focus on incrementalization of structurally recursive functions, and on incrementalizing generally recursive primitives using equational reasoning. We return to this issue in Chapter 20.

15.1.2 Justification

Here, we justify informally the rule for differentiating recursive functions using fixpoint operators.

Let's consider STLC extended with a family of standard fixpoint combinators $fix_\tau : (\tau \to \tau) \to \tau$, with **fix**-reduction defined by equation $\mathbf{fix}\ f \to f\ (\mathbf{fix}\ f)$; we search for a definition of $\mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!]$.

Using informal equational reasoning, if a correct definition of $\mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!]$ exists, it must satisfy

$$\mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!] \cong \mathbf{fix}\ (\mathcal{D}[\![\ f\ ]\!]\ (\mathbf{fix}\ f))$$

We can proceed as follows:

$$\begin{aligned}
& \mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!] \\
= \quad & \{ \text{imposing that } \mathcal{D}[\![-]\!] \text{ respects } \mathbf{fix}\text{-reduction here} \} \\
& \mathcal{D}[\![\ f\ (\mathbf{fix}\ f)\ ]\!] \\
= \quad & \{ \text{using rules for } \mathcal{D}[\![-]\!] \text{ on application} \} \\
& \mathcal{D}[\![\ f\ ]\!]\ (\mathbf{fix}\ f)\ \mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!]
\end{aligned}$$

This is a recursive equation in $\mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!]$, so we can try to solve it using **fix** itself:

[1] The handling of invalid changes is however still ad-hoc; we can use the generic support for sum types, obtain additional equations for cases where the list length changes, and then remove them, with the informal justification that we declared such changes illegal.

$$\mathcal{D}[\![\ \mathbf{fix}\ f\ ]\!] = \mathbf{fix}\ (\lambda dfixf \to \mathcal{D}[\![\ f\ ]\!]\ (\mathbf{fix}\ f)\ dfixf)$$

Indeed, this rule gives a correct derivative. Formalizing our reasoning using denotational semantics would presumably require the use of domain theory. Instead, we prove correct a variant of **fix** in Appendix C, but using operational semantics and step-indexed logical relations.

15.2 Completely invalid changes

In some change sets, some changes might not be valid relative to any source. In particular, we can construct examples in $\Delta(\mathbb{Z} \to \mathbb{Z})$.

To understand why this is plausible, we recall that as described in Sec. 15.3, df can be decomposed into a derivative, and a pointwise function change that is independent of da. While pointwise changes can be defined arbitrarily, the behavior of the derivative of f on changes is determined by the behavior of f.

Example 15.2.1
We search for a function change $df : \Delta(\mathbb{Z} \to \mathbb{Z})$ such that there exist no $f_1, f_2 : \mathbb{Z} \to \mathbb{Z}$ for which $df \triangleright f_1 \hookrightarrow f_2 : \mathbb{Z} \to \mathbb{Z}$. To find df, we assume that there are f_1, f_2 such that $df \triangleright f_1 \hookrightarrow f_2 : \mathbb{Z} \to \mathbb{Z}$, prove a few consequences, and construct df that cannot satisfy them. Alternatively, we could pick the desired definition for df right away, and prove by contradiction that there exist no f_1, f_2 such that $df \triangleright f_1 \hookrightarrow f_2 : \mathbb{Z} \to \mathbb{Z}$.

Recall that on integers $a_1 \oplus da = a_1 + da$, and that $da \triangleright a_1 \hookrightarrow a_2 : \mathbb{Z}$ means $a_2 = a_1 \oplus da = a_1 + da$. So, for any numbers a_1, da, a_2 such that $a_1 + da = a_2$, validity of df implies that

$$f_2\ (a_1 + da) = f_1\ a_1 + df\ a_1\ da.$$

For any two numbers b_1, db such that $b_1 + db = a_1 + da$, we have that

$$f_1\ a_1 + df\ a_1\ da = f_2\ (a_1 + da) = f_2\ (b_1 + db) = f_1\ b_1 + df\ b_1\ db.$$

Rearranging terms, we have

$$df\ a_1\ da - df\ b_1\ db = f_1\ b_1 - f_1\ a_1,$$

that is, $df\ a_1\ da - df\ b_1\ db$ does not depend on da and db.

For concreteness, let us fix $a_1 = 0$, $b_1 = 1$, and $a_1 + da = b_1 + db = s$. We have then that

$$df\ 0\ s - df\ 1\ (s - 1) = f_1\ 1 - f_1\ 0,$$

Once we set $h = f_1\ 1 - f_1\ 0$, we have $df\ 0\ s - df\ 1\ (s - 1) = h$. Because s is just the sum of two arbitrary numbers, while h only depends on f_1, this equation must hold for a fixed h and for all integers s.

To sum up, we assumed for a given df there exists f_1, f_2 such that $df \triangleright f_1 \hookrightarrow f_2 : \mathbb{Z} \to \mathbb{Z}$, and concluded that there exists $h = f_1\ 1 - f_1\ 0$ such that for all s

$$df\ 0\ s - df\ 1\ (s - 1) = h.$$

At this point, we can try concrete families of functions df to obtain a contradiction. Substituting a linear polynomial $df\ a\ da = c_1 \cdot a + c_2 \cdot da$ fails to obtain a contradiction: in fact, we can construct various f_1, f_2 such that $df \triangleright f_1 \hookrightarrow f_2 : \mathbb{Z} \to \mathbb{Z}$. So we try quadratic polynomials: Substituting $df\ a\ da = c \cdot da^2$ succeeds: we have that there is h such that for all integers s

$$c \cdot \left(s^2 - (s-1)^2\right) = h.$$

However, $c \cdot \left(s^2 - (s-1)^2\right) = 2 \cdot c \cdot s - c$ which isn't constant, so there can be no such h. □

15.3 Pointwise function changes

We can also decompose function changes into orthogonal (and possibly easier to understand) concepts.

Consider two functions $f_1, f_2 : A \to B$ and two inputs $a_1, a_2 : A$. The difference between $f_2\ a_2$ and $f_1\ a_1$ is due to changes to both the function and its argument. We can compute the whole change at once via a function change df as $df\ a_1\ da$. Or we can compute separately the effects of the function change and of the argument change. We can account for changes from $f_1\ a_1$ to $f_2\ a_2$ using f_1', a derivative of f_1: $f_1'\ a_1\ da = f_1\ a_2 \ominus f_1\ a_2 = f_1\ (a_1 \oplus da) \ominus f\ a_1$.[2]

We can account for changes from f_1 to f_2 using the *pointwise difference* of two functions, $\nabla f_1 = \lambda(a : A) \to f_2\ a \ominus f_1\ a$; in particular, $f_2\ (a_1 \oplus da) \ominus f_1\ (a_1 \oplus da) = \nabla f\ (a_1 \oplus da)$. Hence, a function change simply *combines* a derivative with a pointwise change using change composition:

$$\begin{aligned} df\ a_1\ da &= f_2\ a_2 \ominus f_1\ a_1 \\ &= (f_1\ a_2 \ominus f_1\ a_1) \circledcirc (f_2\ a_2 \ominus f_1\ a_2) \\ &= f_1'\ a_1\ da \circledcirc \nabla f\ (a_1 \oplus da) \end{aligned} \tag{15.1}$$

One can also compute a pointwise change from a function change:

$$\nabla\ f\ a = df\ a\ \mathbf{0}_a$$

While some might find pointwise changes a more natural concept, we find it easier to use our definitions of function changes, which combines both pointwise changes and derivatives into a single concept. Some related works explore the use of pointwise changes; we discuss them in Sec. 19.2.2.

15.4 Modeling only valid changes

In this section, we contrast briefly the formalization of ILC in this thesis (for short ILC'17) with the one we used in our first formalization [Cai et al., 2014] (for short ILC'14). We keep the discussion somewhat informal; we have sketched proofs of our claims and mechanized some, but we omit all proofs here. We discuss both formalizations using our current notation and terminology, except for concepts that are not present here.

Both formalizations model function changes semantically, but the two models we present are different. Overall, ILC'17 uses simpler machinery and seems easier to extend to more general base languages, and its mechanization of ILC'17 appears simpler and smaller. Instead, ILC'14 studies additional entities but better behaved entities.

In ILC'17, input and output domains of function changes contain *invalid* changes, while in ILC'14 these domains are restricted to valid changes via dependent types; ILC'14 also considers the denotation of $\mathcal{D}[\![\ t\ ]\!]$, whose domains include invalid changes, but such denotations are studied only indirectly. In both cases, function changes must map valid changes to valid changes. But ILC'14, application $df\ v_1\ dv$ is only well-typed is dv is a change valid from v_1, hence we can simply say that $df\ v_1$ respects change equivalence. As discussed in Sec. 14.2, in ILC'17 the analogous property has a trickier statement: we can write $df\ v_1$ and apply it to arbitrary equivalent changes $dv_1 =_\Delta dv_2$, even if their source is not v_1, but such change equivalences are not preserved.

[2] For simplicity, we use equality on changes, even though equality is too restrictive. Later (in Sec. 14.2) we'll define an equivalence relation on changes, called change equivalence and written $=_\Delta$, and use it systematically to relate changes in place of equality. For instance, we'll write that $f_1'\ a_1\ da =_\Delta f_1\ (a_1 \oplus da) \ominus f_1\ a_1$. But for the present discussion, equality will do.

We can relate the two models by defining a logical relation called *erasure* (similar to the one described by Cai et al.): an ILC'14 function change df erases to an ILC'17 function change df' relative to source $f : A \to B$ if, given any change da that erases to da' relative to source $a_1 : A$, output change $df\ a_1\ da$ erases to $df'\ a_1\ da'$ relative to source $f\ a_1$. For base types, erasure simply connects corresponding da (with source) with da' in a manner dependent from the base type (often, just throwing away any embedded proofs of validity). In all cases, one can show that if and only if dv erases to dv' with source v_1, then $v_1 \oplus dv = v_2 \oplus dv'$ (for suitable variants of $\oplus$): in other words, dv and dv' share source and destination (technically, ILC'17 changes have no fixed source, so we say that they are changes from v_1 to v_2 for some v_2).

In ILC'14 there is a different incremental semantics $[\![\ t\ ]\!]^\Delta$ for terms t, but it is still a valid ILC'14 change. One can show that $[\![\ t\ ]\!]^\Delta$ (as defined in ILC'14) erases to $[\![\ \mathcal{D}[\![\ t\ ]\!]\ ]\!]^\Delta$ (as defined in ILC'17) relative to source $[\![\ t\ ]\!]$; in fact, the needed proof is sketched by Cai et al., through in disguise.

It seems clear there is no isomorphism between ILC'14 changes and ILC'17 changes. An ILC'17 function change also accepts invalid changes, and the behavior on those changes can't be preserved by an isomorphism. Worse, it seems hard to define a non-isomorphic mapping: to map an ILC'14 change df to an an ILC'17 change *erase* df, we have to define behavior for (*erase* df) $a\ da$ even when da is invalid. As long as we work in a constructive setting, we cannot decide whether da is valid in general, because da can be a function change with infinite domain.

We can give however a definition that does not need to detect such invalid changes: Just extract source and destination from a function change using valid change $\mathbf{0}_v$, and take difference of source and destination using $\ominus$ in the target system.

$$
\begin{aligned}
&\mathit{unerase}\ (\sigma \to \tau)\ df' = \mathbf{let}\ f = \lambda v \to df'\ v\ \mathbf{0}_v\ \mathbf{in}\ (f \oplus df') \ominus f\\
&\mathit{unerase}\ \_\ dv' = \ldots\\
&\mathit{erase}\ (\sigma \to \tau)\ df = \mathbf{let}\ f = \lambda v \to df\ v\ \mathbf{0}_v\ \mathbf{in}\ (f \oplus df) \ominus f\\
&\mathit{erase}\ \_\ dv = \ldots
\end{aligned}
$$

We define these function by induction on types (for elements of $\Delta\tau$, not arbitrary change structures), and we overload $\ominus$ for ILC'14 and ILC'17. We conjecture that for all types τ and for all ILC'17 changes dv' (of the right type), *unerase* $\tau\ dv'$ erases to dv', and for all ILC'14 changes dv, dv erases to *erase* $\tau\ dv'$.

Erasure is a well-behaved logical relation, similar to the ones relating source and destination language of a compiler and to partial equivalence relations. In particular, it also induces partial equivalence relations (PER) (see Sec. 14.2.3), both on ILC'14 changes and on ILC'17 changes: two ILC'14 changes are equivalent if they erase to the same ILC'17 change, and two ILC'17 changes are equivalent if the same ILC'14 change erases to both. Both relations are partial equivalence relations (and total on valid changes). Because changes that erase to each other share source and destination, these induced equivalences coincide again with change equivalence. That both relations are PERs also means that erasure is a so-called *quasi-PER* [Krishnaswami and Dreyer, 2013]. Quasi-PERs are a natural (though not obvious) generalization of PERs for relations among different sets $R \subseteq S_1 \times S_2$: such relations cannot be either symmetric or transitive. However, we make no use of additional properties of quasi-PERs, hence we don't discuss them in further detail.

15.4.1 One-sided vs two-sided validity

There are also further superficial differences among the two definitions. In ILC'14, changes valid with soure a have dependent type Δa. This dependent type is indexed by the source but not by the destination. Dependent function changes with source $f : A \to B$ have type $(a : A) \to \Delta a \to \Delta(f\ a)$, relating the behavior of function change df with the behavior of f on original inputs. But this is

half of function validity: to relate the behavior of df with the behavior of df on updated inputs, in ILC'14 valid function changes have to satisfy an additional equation called *preservation of future*:[3]

$$f_1\ a_1 \oplus df\ a_1\ da = (f_1 \oplus df)\ (a_1 \oplus da).$$

This equation appears inelegant, and mechanized proofs were often complicated by the need to perform rewritings using it. Worse, to show that a function change is valid, we have to use different approaches to prove it has the correct source and the correct destination.

This difference is however superficial. If we replace $f_1 \oplus df$ with f_2 and $a_1 \oplus da$ with a_2, this equation becomes $f_1\ a_1 \oplus df\ a_1\ da = f_2\ a_2$, a consequence of $f_2 \triangleright df \hookrightarrow - : f_1$. So one might suspect that ILC'17 valid function changes also satisfy this equation. This is indeed the case:

Lemma 15.4.1
A valid function change $df \triangleright f_1 \hookrightarrow f_2 : A \to B$ satisfies equation

$$f_1\ a_1 \oplus df\ a_1\ da = (f_1 \oplus df)\ (a_1 \oplus da)$$

on any valid input $da \triangleright a_1 \hookrightarrow a_2 : A \to B$. □

Conversely, one can also show that ILC'14 function changes also satisfy two-sided validity as defined in ILC'17. Hence, the only true difference between ILC'14 and ILC'17 models is the one we discussed earlier, namely whether function changes can be applied to invalid inputs or not.

We believe it could be possible to formalize the ILC'14 model using two-sided validity, by defining a dependent type of valid changes: $\Delta_2\ (A \to B)\ f_1\ f_2 = (a_1\ a_2 : A) \to \Delta_2\ A\ a_1\ a_2 \to \Delta_2\ B\ (f_1\ a_1)\ (f_2\ a_2)$. We provide more details on such a transformation in Chapter 18.

Models restricted to valid changes (like ILC'14) are related to models based on directed graphs and reflexive graphs, where values are graphs vertexes, changes are edges between change source and change destination (as hinted earlier). In graph language, validity preservation means that function changes are graph homomorphisms.

Based on similar insights, Atkey [2015] suggests modeling ILC using reflexive graphs, which have been used to construct parametric models for System F and extensions, and calls for research on the relation between ILC and parametricity. As follow-up work, Cai [2017] studies models of ILC based on directed and reflexive graphs.

[3] Name suggested by Yufei Cai.

Chapter 16

Differentiation in practice

In practice, successful incrementalization requires both correctness and performance of the derivatives. Correctness of derivatives is guaranteed by the theoretical development the previous sections, together with the interface for differentiation and proof plugins, whereas performance of derivatives has to come from careful design and implementation of differentiation plugins.

16.1 The role of differentiation plugins

Users of our approach need to (1) choose which base types and primitives they need, (2) implement suitable differentiation plugins for these base types and primitives, (3) rewrite (relevant parts of) their programs in terms of these primitives and (4) arrange for their program to be called on changes instead of updated inputs.

As discussed in Sec. 10.5, differentiation supports abstraction, application and variables, but since computation on base types is performed by primitives for those types, efficient derivatives for primitives are essential for good performance.

To make such derivatives efficient, change types must also have efficient implementations, and allow describing precisely what changed. The efficient derivative of *sum* in Chapter 10 is possible only if bag changes can describe deletions and insertions, and integer changes can describe additive differences.

For many conceivable base types, we do not have to design the differentiation plugins from scratch. Instead, we can reuse the large body of existing research on incrementalization in first-order and domain-specific settings. For instance, we reuse the approach from Gluche et al. [1997] to support incremental bags and maps. By wrapping a domain-specific incrementalization result in a differentiation plugin, we adapt it to be usable in the context of a higher-order and general-purpose programming language, and in interaction with other differentiation plugins for the other base types of that language.

For base types with no known incrementalization strategy, the precise interfaces for differentiation and proof plugins can guide the implementation effort. These interfaces could also from the basis for a library of differentiation plugins that work well together.

Rewriting whole programs in our language would be an excessive requirements. Instead, we embed our object language as an EDSL in some more expressive meta-language (Scala in our case study), so that embedded programs are reified. The embedded language can be made to resemble the metalanguage [Rompf and Odersky, 2010]. To incrementalize a part of a computation, we write it in our embedded object language, invoke $\mathcal{D}$ on the embedded program, optionally optimize the resulting programs and finally invoke them. The metalanguage also acts as a macro system for the

```
histogram :: Map Int (Bag word) → Map word Int
histogram = mapReduce groupOnBags additiveGroupOnIntegers histogramMap histogramReduce
  where additiveGroupOnIntegers = Group (+) (λn → −n) 0
        histogramMap _          = foldBag groupOnBags (λn → singletonBag (n, 1))
        histogramReduce _       = foldBag additiveGroupOnIntegers id
-- Precondition:
-- For every key_1 :: k_1 and key_2 :: k_2, the terms mapper key_1 and reducer key_2 are homomorphisms.
mapReduce :: Group v_1 → Group v_3 → (k_1 → v_1 → Bag (k_2, v_2)) → (k_2 → Bag v_2 → v_3) →
             Map k_1 v_1 → Map k_2 v_3
mapReduce group_1 group_3 mapper reducer = reducePerKey ∘ groupByKey ∘ mapPerKey
  where mapPerKey    = foldMap group_1 groupOnBags mapper
        groupByKey   = foldBag  (groupOnMaps groupOnBags)
                                (λ(key, val) → singletonMap key (singletonBag val))
        reducePerKey = foldMap groupOnBags (groupOnMaps group_3)
                                (λkey bag    → singletonMap key (reducer key bag))
```

Figure 16.1: The λ-term *histogram* with Haskell-like syntactic sugar. *additiveGroupOnIntegers* is the abelian group induced on integers by addition $(\mathbb{Z}, +, 0, -)$.

object language, as usual. This allows us to simulate polymorphic collections such as (**Bag** ι) even though the object language is simply-typed; technically, our plugin exposes a family of base types to the object language.

16.2 Predicting nil changes

Handling changes to all inputs can induce excessive overhead in incremental programs [Acar, 2009]. It is also often unnecessary; for instance, the function argument of *fold* in Chapter 10 does not change since it is a closed subterm of the program, so *fold* will receive a nil change for it. A (conservative) static analysis can detect changes that are guaranteed to be nil at runtime. We can then specialize derivatives that receive this change, so that they need not inspect the change at runtime.

For our case study, we have implemented a simple static analysis which detects and propagates information about closed terms. The analysis is not interesting and we omit details for lack of space.

16.2.1 Self-maintainability

In databases, a self-maintainable view [Gupta and Mumick, 1999] is a function that can update its result from input changes alone, without looking at the actual input. By analogy, we call a derivative *self-maintainable* if it uses no base parameters, only their changes. Self-maintainable derivatives describe efficient incremental computations: since they do not use their base input, their running time does not have to depend on the input size.

Example 16.2.1
$\mathcal{D}[\![\, merge \,]\!] = \lambda x\ dx\ y\ dy \to merge\ dx\ dy$ is self-maintainable with the change structure $\widehat{\textbf{Bag}\ S}$ described in Example 10.3.1, because it does not use the base inputs x and y. Other derivatives are self-maintainable only in certain contexts. The derivative of element-wise function application (*map f xs*) ignores the original value of the bag *xs* if the changes to f are always nil, because the underlying primitive *foldBag* is self-maintainable in this case (as discussed in next section). We take advantage of this by implementing a specialized derivative for *foldBag*.

Similarly to what we have seen in Sec. 10.6 that *dgrandTotal* needlessly recomputes *merge xs ys* without optimizations. However, the result is a base input to *fold′*. In next section, we'll replace *fold′* by a self-maintainable derivative (based again on *foldBag*) and will avoid this recomputation.□

To conservatively predict whether a derivative is going to be self-maintainable (and thus efficient), one can inspect whether the program restricts itself to (conditionally) self-maintainable primitives, like *merge* (always) or *map f* (only if *df* is nil, which is guaranteed when *f* is a closed term).

To avoid recomputing base arguments for self-maintainable derivatives (which never need them), we currently employ lazy evaluation. Since we could use standard techniques for dead-code elimination [Appel and Jim, 1997] instead, laziness is not central to our approach.

A significant restriction is that not-self-maintainable derivatives can require expensive computations to supply their base arguments, which can be expensive to compute. Since they are also computed while running the base program, one could reuse the previously computed value through memoization or extensions of static caching (as discussed in Sec. 19.2.3). We leave implementing these optimizations for future work. As a consequence, our current implementation delivers good results only if most derivatives are self-maintainable.

16.3 Case study

We perform a case study on a nontrivial realistic program to demonstrate that ILC can speed it up. We take the MapReduce-based skeleton of the word-count example [Lämmel, 2007]. We define a suitable differentiation plugin, adapt the program to use it and show that incremental computation is faster than recomputation. We designed and implemented the differentiation plugin following the requirements of the corresponding proof plugin, even though we did not formalize the proof plugin (e.g. in Agda). For lack of space, we focus on base types which are crucial for our example and its performance, that is, collections. The plugin also implements tuples, tagged unions, Booleans and integers with the usual introduction and elimination forms, with few optimizations for their derivatives.

wordcount takes a map from document IDs to documents and produces a map from words appearing in the input to the count of their appearances, that is, a histogram:

$$\mathit{wordcount} : \mathbf{Map\ ID\ Document} \to \mathbf{Map\ Word\ Int}$$

For simplicity, instead of modeling strings, we model documents as bags of words and document IDs as integers. Hence, what we implement is:

$$\mathit{histogram} : \mathbf{Map\ Int\ (Bag}\ a) \to \mathbf{Map}\ a\ \mathbf{Int}$$

We model words by integers ($a = \mathit{Int}$), but treat them parametrically. Other than that, we adapt directly Lämmel's code to our language. Figure 16.1 shows the λ-term *histogram*.

Figure 16.2 shows a simplified Scala implementation of the primitives used in Fig. 16.1. As bag primitives, we provide constructors and a fold operation, following Gluche et al. [1997]. The constructors for bags are $\emptyset$ (constructing the empty bag), *singleton* (constructing a bag with one element), *merge* (constructing the merge of two bags) and *negate* (*negate b* constructs a bag with the same elements as *b* but negated multiplicities); all but *singleton* represent abelian group operations. Unlike for usual ADT constructors, the same bag can be constructed in different ways, which are equivalent by the equations defining abelian groups; for instance, since *merge* is commutative, $\mathit{merge}\ x\ y = \mathit{merge}\ y\ x$. Folding on a bag will represent the bag through constructors in an arbitrary way, and then replace constructors with arguments; to ensure a well-defined result, the arguments of fold should respect the same equations, that is, they should form an abelian group; for instance, the binary operator should be commutative. Hence, the fold operator *foldBag* can be defined to take a function (corresponding to *singleton*) and an abelian group (for the other constructors). *foldBag* is

```
// Abelian groups
abstract class Group[A] {
  def merge( value₁ : A,  value₂ : A): A
  def inverse(value: A): A
  def zero: A
}

// Bags
type Bag[A] = collection.immutable.HashMap[A, Int]

def groupOnBags[A] = new Group[Bag[A]] {
  def merge( bag₁ : Bag[A],  bag₂ : Bag[A]) = ...
  def inverse(bag: Bag[A]) = bag.map({
    case (value, count) ⇒ (value, -count)
  })
  def zero = collection.immutable.HashMap()
}

def foldBag[A, B](group: Group[B], f: A ⇒ B, bag: Bag[A]): B =
  bag.flatMap({
    case (x, c) if c ≥ 0 ⇒ Seq.fill(c)(f(x))
    case (x, c) if c < 0 ⇒ Seq.fill(-c)(group.inverse(f(x)))
  }).fold(group.zero)(group.merge)

// Maps
type Map[K, A] = collection.immutable.HashMap[K, A]

def groupOnMaps[K, A](group: Group[A]) = new Group[Map[K, A]] {
  def merge( dict₁ : Map[K, A],  dict₂ : Map[K, A]): Map[K, A] =
    dict₁ .merged( dict₂ )({
      case ((k, v₁), (_, v₂)) ⇒ (k, group.merge(v₁, v₂))
    }).filter({
      case (k, v) ⇒ v ≠ group.zero
    })

  def inverse(dict: Map[K, A]): Map[K, A] = dict.map({
    case (k, v) ⇒ (k, group.inverse(v))
  })

  def zero = collection.immutable.HashMap()
}

// The general map fold
def foldMapGen[K, A, B](zero: B, merge: (B, B) ⇒ B)
  (f: (K, A) ⇒ B, dict: Map[K, A]): B =
    dict.map(Function.tupled(f)).fold(zero)(merge)

// By using foldMap instead of foldMapGen, the user promises that
// f k is a homomorphism from groupA to groupB for each k : K.
def foldMap[K, A, B](groupA: Group[A], groupB: Group[B])
  (f: (K, A) ⇒ B, dict: Map[K, A]): B =
    foldMapGen(groupB.zero, groupB.merge)(f, dict)
```

Figure 16.2: A Scala implementation of primitives for bags and maps. In the code, we call ⊞, ⊟ and e respectively *merge*, *inverse*, and *zero*. We also omit the relatively standard primitives.

then defined by equations:

$$\begin{aligned}
&\mathit{foldBag} : \mathbf{Group}\ \tau \to (\sigma \to \tau) \to && \mathbf{Bag}\ \sigma \to \tau \\
&\mathit{foldBag}\ g@(_, \boxplus, \boxminus, e)\ f\ \emptyset && = e \\
&\mathit{foldBag}\ g@(_, \boxplus, \boxminus, e)\ f\ (\mathit{merge}\ b_1\ b_2) && = \mathit{foldBag}\ g\ f\ b_1 \\
& && \quad \boxplus\ \mathit{foldBag}\ g\ f\ b_1 \\
&\mathit{foldBag}\ g@(_, \boxplus, \boxminus, e)\ f\ (\mathit{negate}\ b) && = \boxminus\ (\mathit{foldBag}\ g\ f\ b) \\
&\mathit{foldBag}\ g@(_, \boxplus, \boxminus, e)\ f\ (\mathit{singleton}\ v) && = f\ v
\end{aligned}$$

If g is a group, these equations specify *foldBag* g precisely [Gluche et al., 1997]. Moreover, the first three equations mean that *foldBag* $g\ f$ is the *abelian group homomorphism* between the abelian group on bags and the group g (because those equations coincide with the definition). Figure 16.2 shows an implementation of *foldBag* as specified above. Moreover, all functions which deconstruct a bag can be expressed in terms of *foldBag* with suitable arguments. For instance, we can sum the elements of a bag of integers with *foldBag* $gZ\ (\lambda x \to x)$, where gZ is the abelian group induced on integers by addition $(\mathbb{Z}, +, 0, -)$. Users of *foldBag* can define different abelian groups to specify different operations (for instance, to multiply floating-point numbers).

If g and f do not change, *foldBag* $g\ f$ has a self-maintainable derivative. By the equations above,

$$\begin{aligned}
&\mathit{foldBag}\ g\ f\ (b \oplus db) \\
&= \mathit{foldBag}\ g\ f (\mathit{merge}\ b\ db) \\
&= \mathit{foldBag}\ g\ f\ b \boxplus \mathit{foldBag}\ g\ f\ db \\
&= \mathit{foldBag}\ g\ f\ b \oplus \mathit{GroupChange}\ g\ (\mathit{foldBag}\ g\ f\ db)
\end{aligned}$$

We will describe the *GroupChange* change constructor in a moment. Before that, we note that as a consequence, the derivative of *foldBag* $g\ f$ is

$$\lambda b\ db \to \mathit{GroupChange}\ g\ (\mathit{foldBag}\ g\ f\ db)\,,$$

and we can see it does not use b: as desired, it is *self-maintainable*. Additional restrictions are require to make *foldMap*'s derivative self-maintainable. Those restrictions require the precondition on *mapReduce* in Fig. 16.1. *foldMapGen* has the same implementation but without those restrictions; as a consequence, its derivative is not self-maintainable, but it is more generally applicable. Lack of space prevents us from giving more details.

To define *GroupChange*, we need a suitable erased change structure on τ, such that $\oplus$ will be equivalent to $\boxplus$. Since there might be multiple groups on τ, we *allow the changes to specify a group*, and have $\oplus$ delegate to $\boxplus$ (which we extract by pattern-matching on the group):

$$\begin{aligned}
&\Delta\tau = \mathit{Replace}\ \tau \mid \mathit{GroupChange}\,(\mathbf{AbelianGroup}\ \tau)\,\tau \\
&v \oplus (\mathit{Replace}\ u) = u \\
&v \oplus (\mathit{GroupChange}\ (\boxplus, \mathit{inverse}, \mathit{zero})\ dv) = v \boxplus dv \\
&v \ominus u = \mathit{Replace}\ v
\end{aligned}$$

That is, a change between two values is either simply the new value (which replaces the old one, triggering recomputation), or their difference (computed with abelian group operations, like in the changes structures for groups from Sec. 13.1.1. The operator $\ominus$ does not know which group to use, so it does not take advantage of the group structure. However, *foldBag* is now able to generate a group change.

We rewrite *grandTotal* in terms of *foldBag* to take advantage of group-based changes.

$$
\begin{aligned}
&id = \lambda x \to x \\
&G_+ = (\mathbb{Z}, +, -, 0) \\
&grandTotal = \lambda xs \to \lambda ys \to foldBag\ G_+\ id\ (merge\ xs\ ys) \\
&\mathcal{D}[\![\ grandTotal\ ]\!] = \\
&\quad \lambda xs \to \lambda dxs \to \lambda ys \to \lambda dys \to \\
&\qquad foldBag'\ G_+\ G'_+\ id\ id' \\
&\qquad\quad (merge\ xs\ ys) \\
&\qquad\quad (merge'\ xs\ dxs\ ys\ dys)
\end{aligned}
$$

It is now possible to write down the derivative of *foldBag*.

$$
\begin{aligned}
&\text{(if static analysis detects that } dG \text{ and } df \text{ are nil changes)} \\
&foldBag' = \mathcal{D}[\![\ foldBag\ ]\!] = \\
&\quad \lambda G \to \lambda dG \to \lambda f \to \lambda df \to \lambda zs \to \lambda dzs \to \\
&\qquad GroupChange\ G\ (foldBag\ G\ f\ dzs)
\end{aligned}
$$

We know from Sec. 10.6 that

$$merge' = \lambda u \to \lambda du \to \lambda v \to \lambda dv \to merge\ du\ dv.$$

Inlining *foldBag*′ and *merge*′ gives us a more readable term β-equivalent to the derivative of *grandTotal*:

$$
\begin{aligned}
&\mathcal{D}[\![\ grandTotal\ ]\!] = \\
&\quad \lambda xs \to \lambda dxs \to \lambda ys \to \lambda dys \to foldBag\ G_+\ id\ (merge\ dxs\ dys).
\end{aligned}
$$

16.4 Benchmarking setup

We run object language programs by generating corresponding Scala code. To ensure rigorous benchmarking [Georges et al., 2007], we use the Scalameter benchmarking library. To show that the performance difference from the baseline is statistically significant, we show 99%-confidence intervals in graphs.

We verify Eq. (10.1) experimentally by checking that the two sides of the equation always evaluate to the same value.

We ran benchmarks on an 8-core Intel Core i7-2600 CPU running at 3.4 GHz, with 8GB of RAM, running Scientific Linux release 6.4. While the benchmark code is single-threaded, the JVM offloads garbage collection to other cores. We use the preinstalled OpenJDK 1.7.0_25 and Scala 2.10.2.

Input generation Inputs are randomly generated to resemble English words over all webpages on the internet: The vocabulary size and the average length of a webpage stay relatively the same, while the number of webpages grows day by day. To generate a size-n input of type (**Map Int** (**Bag Int**)), we generate n random numbers between 1 and 1000 and distribute them randomly in $n/1000$ bags. Changes are randomly generated to resemble edits. A change has 50% probability to delete a random existing number, and has 50% probability to insert a random number at a random location.

Experimental units Thanks to Eq. (10.1), both recomputation $f\ (a \oplus da)$ and incremental computation $f\ a \oplus \mathcal{D}[\![f]\!]\ a\ da$ produce the same result. Both programs are written in our object language. To show that derivatives are faster, we compare these two computations. To compare with recomputation, we measure the *aggregated* running time for running the derivative on the change and for updating the original output with the result of the derivative.

16.5 Benchmark results

Our results show (Fig. 16.3) that our program reacts to input changes in essentially constant time, as expected, hence orders of magnitude faster than recomputation. Constant factors are small enough that the speedup is apparent on realistic input sizes.

We present our results in Fig. 16.3. As expected, the runtime of incremental computation is *essentially constant* in the size of the input, while the runtime of recomputation is linear in the input size. Hence, on our biggest inputs incremental computation is over 10^4 times faster.

Derivative time is in fact slightly irregular for the first few inputs, but this irregularity decreases slowly with increasing warmup cycles. In the end, for derivatives we use 10^4 warmup cycles. With fewer warmup cycles, running time for derivatives decreases significantly during execution, going from 2.6ms for $n = 1000$ to 0.2ms for $n = 512000$. Hence, we believe extended warmup is appropriate, and the changes do not affect our general conclusions. Considering confidence intervals, in our experiments the running time for derivatives varies between 0.139ms and 0.378ms.

In our current implementation, the code of the generated derivatives can become quite big. For the histogram example (which is around 1KB of code), a pretty-print of its derivative is around 40KB of code. The function application case in Fig. 12.1c can lead to a quadratic growth in the worst case. More importantly, we realized that our home-grown transformation system in some cases performs overly aggressive inlining, especially for derivatives, even though this is not required for incrementalization, and believe this explains a significant part of the problem. Indeed, code blowup problems do not currently appear in later experiments (see Sec. 17.4).

16.6 Chapter conclusion

Our results show that the incrementalized program runs in essentially constant time and hence orders of magnitude faster than the alternative of recomputation from scratch.

An important lessons from the evaluations is that, as anticipated in Sec. 16.2.1, to achieve good performance our current implementation requires some form of dead code elimination, such as laziness.

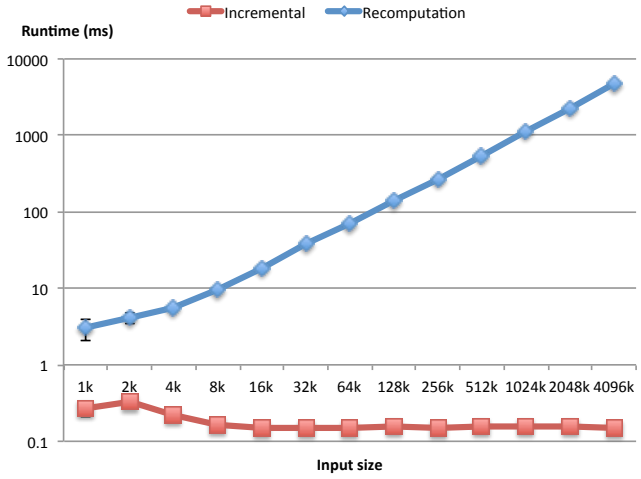


Figure 16.3: Performance results in log–log scale, with input size on the x-axis and runtime in ms on the y-axis. Confidence intervals are shown by the whiskers; most whiskers are too small to be visible.

Chapter 17

Cache-transfer-style conversion

17.1 Introduction

Incremental computation is often desirable: after computing an output from some input, we often need to produce new outputs corresponding to changed inputs. One can simply rerun the same *base program* on the new input; but instead, incremental computation transforms the input change to an output change. This can be desirable because more efficient.

Incremental computation could also be desirable because the changes themselves are of interest: Imagine a typechecker explaining how some change to input source propagates to changes to the typechecking result. More generally, imagine a program explaining how some change to its input propagates through a computation into changes to the output.

ILC (Incremental λ-Calculus) [Cai et al., 2014] is a recent approach to incremental computation for higher-order languages. ILC represents changes from an old value v_1 to an updated value v_2 as a first-class value written dv. ILC also transforms statically *base programs* to *incremental programs* or *derivatives*: derivatives produce output changes from changes to all their inputs. Since functions are first-class values, ILC introduces a notion of function changes.

However, as mentioned by Cai et al. and as we explain below, ILC as currently defined does not allow reusing intermediate results created by the base computation during the incremental computation. That restricts ILC to supporting efficiently only *self-maintainable computations*, a rather restrictive class: for instance, mapping self-maintainable functions on a sequence is self-maintainable, but dividing numbers isn't! In this paper, we remove this limitation.

To remember intermediate results, many incrementalization approaches rely on forms of memoization: one uses hashtables to memoize function results, or dynamic dependence graphs [Acar, 2005] to remember the computation trace. However, such data structures often remember results that might not be reused; moreover, the data structures themselves (as opposed to their values) occupy additional memory, looking up intermediate results has a cost in time, and typical general-purpose optimizers cannot predict results from memoization lookups. Instead, ILC aims to produce purely functional programs that are suitable for further optimizations.

We eschew memoization: instead, we transform programs to *cache-transfer style (CTS)*, following ideas from Liu and Teitelbaum [1995]. CTS functions output *caches* of intermediate results along with their primary results. Caches are just nested tuples whose structure is derived from code, and accessing them does not involve looking up keys depending on inputs. We also extend differentiation to produce *CTS derivatives*, which can extract from caches any intermediate results they need. This approach was inspired and pioneered by Liu and Teitelbaum for untyped first-order functional languages, but we integrate it with ILC and extend it to higher-order typed languages.

While CTS functions still produce additional intermediate data structures, produced programs can be subject to further optimizations. We believe static analysis of a CTS function and its CTS derivative can identify and remove unneeded state (similar to what has been done by Liu and Teitelbaum), as we discuss in Sec. 17.5.6. We leave a more careful analysis to future work.

We prove most of our formalization correct in Coq To support non-simply-typed programs, all our proofs are for untyped λ-calculus, while previous ILC correctness proofs were restricted to simply-typed λ-calculus. Unlike previous ILC proofs, we simply define which changes are valid via a *logical relation*, then show the fundamental property for this logical relation (see Sec. 17.2.1). To extend this proof to untyped λ-calculus, we switch to *step-indexed* logical relations.

To support differentiation on our case studies, we also represent function changes as closures that can be inspected, to support manipulating them more efficiently and detecting at runtime when a function change is *nil* hence need not be applied. To show this representation is correct, we also use closures in our mechanized proof.

Unlike plain ILC, typing programs in CTS is challenging, because the shape of caches for a function depends on the function implementation. Our case studies show how to non-trivially embed resulting programs in typed languages, at least for our case studies, but our proofs support an untyped target language.

In sum, we present the following contributions:

- via examples, we motivate extending ILC to remember intermediate results (Sec. 17.2);
- we give a novel proof of correctness for ILC for untyped λ-calculus, based on step-indexed logical relations (Sec. 17.3.3);
- building on top of ILC-style differentiation, we show how to transform untyped higher-order programs to *cache-transfer-style* (CTS) (Sec. 17.3.5);
- we show that programs and derivatives in cache-transfer style *simulate* correctly their non-CTS variants (Sec. 17.3.6);
- we mechanize in Coq most of our proofs;
- we perform performance case studies (in Sec. 17.4) applying (by hand) extension of this technique to Haskell programs, and incrementalize efficiently also programs that do not admit self-maintainable derivatives.

The rest of the paper is organized as follows. Sec. 17.2 summarizes ILC and motivates the extension to cache-transfer style. Sec. 17.3 presents our formalization and proofs. Sec. 17.4 presents our case studies and benchmarks. Sec. 17.5 discusses limitations and future work. Sec. 17.6 discusses related work and Sec. 17.7 concludes.

17.2 Introducing Cache-Transfer Style

In this section we motivate cache-transfer style (CTS). Sec. 17.2.1 summarizes a reformulation of ILC, so we recommend it also to readers familiar with Cai et al. [2014]. In Sec. 17.2.2 we consider a minimal first-order example, namely an average function. We incrementalize it using ILC, explain why the result is inefficient, and show that remembering results via cache-transfer style enables efficient incrementalization with asymptotic speedups. In Sec. 17.2.3 we consider a higher-order example that requires not just cache-transfer style but also efficient support for both nil and non-nil function changes, together with the ability to detect nil changes.

17.2.1 Background

ILC considers simply-typed programs, and assumes that base types and primitives come with support for incrementalization.

The ILC framework describes changes as first-class values, and types them using dependent types. To each type A we associate a type ΔA of changes for A, and an *update operator* $\oplus :: A \to \Delta A \to A$, that updates an initial value with a change to compute an updated value. We also consider changes for evaluation environments, which contain changes for each environment entry.

A change $da :: \Delta A$ can be *valid* from $a_1 :: A$ to $a_2 :: A$, and we write then $da \triangleright a_1 \hookrightarrow a_2 :: A$. Then a_1 is the *source* or *initial value* for da, and a_2 is the *destination* or *updated value* for da. From $da \triangleright a_1 \hookrightarrow a_2 :: A$ follows that a_2 coincides with $a_1 \oplus da$, but validity imposes additional invariants that are useful during incrementalization. A change can be valid for more than one source, but a change da and a source a_1 uniquely determine the destination $a_2 = a_1 \oplus da$. To avoid ambiguity, we always consider a change together with a specific source.

Each type comes with its definition of validity: Validity is a ternary *logical relation*. For function types $A \to B$, we define $\Delta(A \to B) = A \to \Delta A \to \Delta B$, and say that a function change $df :: \Delta(A \to B)$ is valid from $f_1 :: A \to B$ to $f_2 :: A \to B$ (that is, $df \triangleright f_1 \hookrightarrow f_2 :: A \to B$) if and only if df maps valid input changes to valid output changes; by that, we mean that if $da \triangleright a_1 \hookrightarrow a_2 :: A$, then $df\ a_1\ da \triangleright f_1\ a_1 \hookrightarrow f_2\ a_2 :: B$. Source and destination of $df\ a_1\ da$, that is $f_1\ a_1$ and $f_2\ a_2$, are the result of applying two different functions, that is f_1 and f_2.

ILC expresses incremental programs as *derivatives*. Generalizing previous usage, we simply say derivative for all terms produced by differentiation. If dE is a valid environment change from E_1 to E_2, and term t is well-typed and can be evaluated against environments E_1, E_2, then term $\mathcal{D}'[\![\ t\ ]\!]$, the derivative of t, evaluated against dE, produces a change from v_1 to v_2, where v_1 is the value of t in environment E_1, and v_2 is the value of t in environment E_2. This correctness property follows immediately the *fundamental property* for the logical relation of validity and can be proven accordingly; we give a step-indexed variant of this proof in Sec. 17.3.3. If t is a function and dE is a nil change (that is, its source E_1 and destination E_2 coincide), then $\mathcal{D}'[\![\ t\ ]\!]$ produces a nil function change and is also a derivative according to Cai et al. [2014].

To support incrementalization, one must define change types and validity for each base type, and a correct derivative for each primitive. Functions written in terms of primitives can be differentiated automatically. As in all approaches to incrementalization (see Sec. 17.6), one cannot incrementalize efficiently an arbitrary program: ILC limits the effort to base types and primitives.

17.2.2 A first-order motivating example: computing averages

Suppose we need to compute the average y of a bag of numbers $xs :: Bag\ \mathbb{Z}$, and that whenever we receive a change $dxs :: \Delta(Bag\ \mathbb{Z})$ to this bag we need to compute the change dy to the average y.

In fact, we expect multiple consecutive updates to our bag. Hence, we say we have an initial bag xs_1 and compute its average y_1 as $y_1 = avg\ xs_1$, and then consecutive changes $dxs_1, dxs_2, \ldots$. Change dxs_1 goes from xs_1 to xs_2, dxs_2 goes from xs_2 to xs_3, and so on. We need to compute $y_2 = avg\ xs_2$, $y_3 = avg\ xs_3$, but more quickly than we would by calling avg again.

We can compute the average through the following function (that we present in Haskell):

```
avg xs =
  let s = sum xs
      n = length xs
      r = div s n
  in r
```

We write this function in *A'-normal form* (A'NF), a small variant of *A-normal form* (ANF) Sabry and Felleisen [1993] that we introduce. In A'NF, programs bind to a variable the result of each function call in *avg*, instead of using it directly; unlike plain ANF, A'NF also binds the result of tail calls such as *div s n* in *avg*. A'NF simplifies conversion to cache-transfer style.

We can incrementalize efficiently both *sum* and *length* by generating via ILC their derivatives *dsum* and *dlength*, assuming a language plugin supporting bags supporting folds.

But division is more challenging. To be sure, we can write the following derivative:

$$
\begin{aligned}
&\mathit{ddiv}\ a_1\ \mathit{da}\ b_1\ \mathit{db} = \\
&\quad \textbf{let}\ a_2 = a_1 \oplus \mathit{da} \\
&\qquad\ \ b_2 = b_1 \oplus \mathit{db} \\
&\quad \textbf{in}\ \ \mathit{div}\ a_2\ b_2 \ominus \mathit{div}\ a_1\ b_1
\end{aligned}
$$

Function *ddiv* computes the difference between the updated and the original result without any special optimization, but still takes $O(1)$ for machine integers. But unlike other derivatives, *ddiv* uses its base inputs a_1 and b_1, that is, it is not *self-maintainable* [Cai et al., 2014].

Because *ddiv* is not self-maintainable, a derivative calling it will not be efficient. To wit, let us look at *davg*, the derivative of *avg*:

$$
\begin{aligned}
&\mathit{davg}\ \mathit{xs}\ \mathit{dxs} = \\
&\quad \textbf{let}\ s\ \ = \mathit{sum}\ \mathit{xs} \\
&\qquad\ \ \mathit{ds} = \mathit{dsum}\ \mathit{xs}\ \mathit{dxs} \\
&\qquad\ \ n\ \ = \mathit{length}\ \mathit{xs} \\
&\qquad\ \ \mathit{dn} = \mathit{dlength}\ \mathit{xs}\ \mathit{dxs} \\
&\qquad\ \ r\ \ = \mathit{div}\ s\ n \\
&\qquad\ \ \mathit{dr} = \mathit{ddiv}\ s\ \mathit{ds}\ n\ \mathit{dn} \\
&\quad \textbf{in}\ \ \mathit{dr}
\end{aligned}
$$

This function recomputes *s*, *n* and *r* just like in *avg*, but *r* is not used so its recomputation can easily be removed by later optimizations. On the other hand, derivative *ddiv* will use the values of its base inputs a_1 and b_1, so derivative *davg* will need to recompute *s* and *n* and save no time over recomputation! If *ddiv* were instead a *self-maintainable* derivative, the computations of *s* and *n* would also be unneeded and could be optimized away. Cai et al. leave efficient support for non-self-maintainable derivatives for future work.

To avoid recomputation we must simply remember intermediate results as needed. Executing $\mathit{davg}\ \mathit{xs}_1\ \mathit{dxs}_1$ will compute exactly the same *s* and *n* as executing $\mathit{avg}\ \mathit{xs}_1$, so we must just save and reuse them, without needing to use memoization proper. Hence, we CTS-convert each function *f* to a *CTS function fC* and a *CTS derivative dfC*: CTS function *fC* produces, together with its final result, a *cache*, that the caller must pass to CTS derivative *dfC*. A cache is just a tuple of values, containing information from subcalls — either inputs (as we explain in a bit), intermediate results or *subcaches*, that is caches returned from further function calls. In fact, typically only primitive functions like *div* need to recall actual result; automatically transformed functions only need to remember subcaches or inputs.

CTS conversion is simplified by first converting to A'NF, where all results of subcomputations are bound to variables: we just collect all caches in scope and return them.

As an additional step, we avoid always passing base inputs to derivatives by defining $\Delta(A \to B) = \Delta A \to \Delta B$. Instead of always passing a base input and possibly not using it, we can simply assume that primitives whose derivative needs a base input store the input in the cache.

To make the translation uniform, we stipulate all functions in the program are transformed to CTS, using a (potentially empty) cache. Since the type of the cache for a function $f :: A \to B$

depends on implementation of f, we introduce for each function f a type for its cache FC, so that CTS function fC has type $A \to (B, FC)$ and CTS derivative dfC has type $\Delta A \to FC \to (\Delta B, FC)$.

The definition of FC is only needed inside fC and dfC, and it can be hidden in other functions to keep implementation details hidden in transformed code; because of limitations of Haskell modules, we can only hide such definitions from functions in other modules.

Since functions of the same type translate to functions of different types, the translation does not preserve well-typedness in a higher-order language in general, but it works well in our case studies (Sec. 17.4); Sec. 17.4.1 shows how to map such functions. We return to this point briefly in Sec. 17.5.1.

CTS-converting our example produces the following code:

```
data AvgC = AvgC SumC LengthC DivC
avgC :: Bag ℤ → (ℤ, AvgC)
avgC xs =
  let (s, cs1)  = sumC xs
      (n, cn1) = lengthC xs
      (r, cr1)  = s `divC` n
  in  (r, AvgC cs1 cn1 cr1)
davgC :: Δ(Bag ℤ) → AvgC → (Δℤ, AvgC)
davgC dxs (AvgC cs1 cn1 cr1) =
  let (ds, cs2)  = dsumC dxs cs1
      (dn, cn2) = dlengthC dxs cn1
      (dr, cr2)  = ddivC ds dn cr1
      in (dr, AvgC cs2 cn2 cr2)
```

In the above program, *sumC* and *lengthC* are self-maintainable, that is they need no base inputs and can be transformed to use empty caches. On the other hand, *ddiv* is not self-maintainable, so we transform it to remember and use its base inputs.

```
divC a b = (a `div` b, (a, b))
ddivC da db (a1, b1) =
  let a2 = a1 ⊕ da
      b2 = b1 ⊕ db
  in  (div a2 b2 ⊖ div a1 b1, (a2, b2))
```

Crucially, *ddivC* must return an updated cache to ensure correct incrementalization, so that application of further changes works correctly. In general, if $(y, c_1) = fC\ x$ and $(dy, c_2) = dfC\ dx\ c_1$, then $(y \oplus dy, c_2)$ must equal the result of the base function fC applied to the new input $x \oplus dx$, that is $(y \oplus dy, c_2) = fC\ (x \oplus dx)$.

Finally, to use these functions, we need to adapt the caller. Let's first show how to deal with a sequence of changes: imagine that dxs_1 is a valid change for xs, and that dxs_2 is a valid change for $xs \oplus dxs_1$. To update the average for both changes, we can then call the *avgC* and *davgC* as follows:

```
  -- A simple example caller with two consecutive changes
avgDAvgC :: Bag ℤ → Δ(Bag ℤ) → Δ(Bag ℤ) →
  (ℤ, Δℤ, Δℤ, AvgC)
avgDAvgC xs dxs1 dxs2 =
  let (res1,   cache1) = avgC   xs
      (dres1, cache2) = davgC dxs1 cache1
      (dres2, cache3) = davgC dxs2 cache2
  in  (res1, dres1, dres2, cache3)
```

Incrementalization guarantees that the produced changes update the output correctly in response to the input changes: that is, we have $res_1 \oplus dres_1 = avg\ (xs \oplus dxs_1)$ and $res_1 \oplus dres_1 \oplus dres_2 = avg\ (xs \oplus dxs_1 \oplus dxs_2)$. We also return the last cache to allow further updates to be processed.

Alternatively, we can try writing a caller that gets an initial input and a (lazy) list of changes, does incremental computation, and prints updated outputs:

```
processChangeList (dxsN : dxss) yN cacheN = do
  let (dy, cacheN') = avg' dxsN cacheN
      yN' = yN ⊕ dy
  print yN'
  processChangeList dxss yN' cacheN'
  -- Example caller with multiple consecutive
  -- changes
someCaller xs1 dxss = do
  let (y1, cache1) = avgC xs1
  processChangeList dxss y1 cache1
```

More in general, we produce both an augmented base function and a derivative, where the augmented base function communicates with the derivative by returning a cache. The contents of this cache are determined statically, and can be accessed by tuple projections without dynamic lookups. In the rest of the paper, we use the above idea to develop a correct transformation that allows incrementalizing programs using cache-transfer style.

We'll return to this example in Sec. 17.4.1.

17.2.3 A higher-order motivating example: nested loops

Next, we consider CTS differentiation on a minimal higher-order example. To incrementalize this example, we enable detecting nil function changes at runtime by representing function changes as closures that can be inspected by incremental programs. We'll return to this example in Sec. 17.4.2.

We take an example of nested loops over sequences, implemented in terms of standard Haskell functions *map* and *concat*. For simplicity, we compute the Cartesian product of inputs:

```
cartesianProduct :: Sequence a → Sequence b → Sequence (a, b)
cartesianProduct xs ys =
  concatMap (λx → map (λy → (x, y)) ys) xs
concatMap f xs = concat (map f xs)
```

Computing *cartesianProduct xs ys* loops over each element x from sequence xs and y from sequence ys, and produces a list of pairs (x, y), taking quadratic time $O(n^2)$ (we assume for simplicity that $|xs|$ and $|ys|$ are both $O(n)$). Adding a fresh element to either xs or ys generates an output change containing $\Theta(n)$ fresh pairs: hence derivative *dcartesianProduct* must take at least linear time. Thanks to specialized derivatives *dmap* and *dconcat* for primitives *map* and *concat*, *dcartesianProduct* has asymptotically optimal time complexity. To achieve this complexity, *dmap f df* must detect when *df* is a nil function change and avoid applying it to unchanged input elements.

To simplify the transformations we describe, we λ-lift programs before differentiating and transforming them.

```
cartesianProduct xs ys =
  concatMap (mapPair ys) xs
mapPair ys = λx → map (pair x) ys
```

```
pair x = λy → (x, y)
concatMap f xs =
  let yss = map f xs
  in  concat yss
```

Suppose we add an element to either *xs* or *ys*. If change *dys* adds one element to *ys*, then *dmapPair ys dys*, the argument to *dconcatMap*, is a non-nil function change taking constant time, so *dconcatMap* must apply it to each element of $xs \oplus dxs$.

Suppose next that change *dxs* adds one element to *xs* and *dys* is a nil change for *ys*. Then *dmapPair ys dys* is a nil function change. And we must detect this dynamically. If a function change $df :: \Delta(A \to B)$ is represented as a function, and A is infinite, one cannot detect dynamically that it is a nil change. To enable runtime nil change detection, we apply closure conversion on function changes: a function change df, represented as a closure is nil for f only if all environment changes it contains are also nil, and if the contained function is a derivative of f's function.

17.3 Formalization

In this section, we formalize cache-transfer-style (CTS) differentiation and formally prove its soundness with respect to differentiation. We furthermore present a novel proof of correctness for differentiation itself.

To simplify the proof, we encode many invariants of input and output programs by defining input and output languages with a suitably restricted abstract syntax: Restricting the input language simplifies the transformations, while restricting the output languages simplifies the semantics. Since base programs and derivatives have different shapes, we define different syntactic categories of *base terms* and *change terms*.

We define and prove sound three transformations, across two languages: first, we present a variant of differentiation (as defined by Cai et al. [2014]), going from base terms of language λ_{AL} to change terms of λ_{AL}, and we prove it sound with respect to non-incremental evaluation. Second, we define CTS conversion as a pair of transformations going from base terms of λ_{AL}: CTS translation produces CTS versions of base functions as base terms in $i\lambda_{AL}$, and CTS differentiation produces CTS derivatives as change terms in $i\lambda_{AL}$.

As source language for CTS differentiation, we use a core language named λ_{AL}. This language is a common target of a compiler front-end for a functional programming language: in this language, terms are written in λ-lifted and A'-normal form (A'NF), so that every intermediate result is named, and can thus be stored in a cache by CTS conversion and reused later (as described in Sec. 17.2).

The target language for CTS differentiation is named $i\lambda_{AL}$. Programs in this target language are also λ-lifted and in A'NF. But additionally, in these programs every toplevel function f produces a cache which is to be conveyed to the derivatives of f.

The rest of this section is organized as follows. Sec. 17.3.1 presents syntax and semantics of source language λ_{AL}. Sec. 17.3.2 defines differentiation, and Sec. 17.3.3 proves it correct. Sec. 17.3.4 presents syntax and semantics of target language λ_{AL}. Sec. 17.3.5 define CTS conversion, and Sec. 17.3.6 proves it correct. Most of the formalization has been mechanized in Coq (the proofs of some straightforward lemmas are left for future work).

17.3.1 The source language λ_{AL}

Syntax The syntax of λ_{AL} is given in Figure 17.1. Our source language allows representing both *base* terms t and *change* terms dt.

		Base terms	
t	::=	$\textbf{let } y = f\ x \textbf{ in } t$	*Call*
	|	$\textbf{let } y = (\overline{x}) \textbf{ in } t$	*Tuple*
	|	x	*Result*
		Change terms	
dt	::=	$\textbf{let } y = f\ x,\ dy = df\ x\ dx \textbf{ in } dt$	*Call*
	|	$\textbf{let } y = (\overline{x}),\ dy = (\overline{dx}) \textbf{ in } dt$	*Tuple*
	|	dx	*Result*
		Closed values	
v	::=	$E[\lambda x.\ t]$	*Closure*
	|	$(\overline{v})$	*Tuple*
	|	ℓ	*Literal*
	|	$\mathbf{p}$	*Primitive*

		Value environments	
E	::=	$E; x = v$	*Value binding*
	|	$\bullet$	*Empty*
		Change values	
dv	::=	$dE[\lambda x\ dx.\ dt]$	*Closure*
	|	$(\overline{dv})$	*Tuple*
	|	$d\ell$	*Literal*
	|	$!v$	*Replace*
	|	$\textbf{nil}$	*Nil*
		Change environments	
dE	::=	$dE; x = v, dx = dv$	*Binding*
	|	$\bullet$	*Empty*
n, k	$\in$	$\mathbb{N}$	***Step indexes***

Figure 17.1: Source language λ_{AL} (syntax).

$$\frac{}{E \vdash x \Downarrow_1 E(x)}\ [\textsc{SVar}] \qquad \frac{E; y = (E(\overline{x})) \vdash t \Downarrow_n v}{E \vdash \textbf{let } y = (\overline{x}) \textbf{ in } t \Downarrow_{n+1} v}\ [\textsc{STuple}]$$

$$\frac{E(f) = \mathbf{p} \qquad E; y = \delta_{\mathbf{p}}(E(x)) \vdash t \Downarrow_n v}{E \vdash \textbf{let } y = f\ x \textbf{ in } t \Downarrow_{n+1} v}\ [\textsc{SPrimitiveCall}]$$

$$\frac{E(f) = E_f[\lambda x.\ t_f] \qquad E_f; x = E(x) \vdash t_f \Downarrow_m v_y \qquad E; y = v_y \vdash t \Downarrow_n v}{E \vdash \textbf{let } y = f\ x \textbf{ in } t \Downarrow_{m+n+1} v}\ [\textsc{SClosureCall}]$$

Figure 17.2: Step-indexed big-step semantics for base terms of source language λ_{AL}.

Our syntax for base terms t represents λ-lifted programs in A'-normal form (Sec. 17.2.2). We write $\overline{x}$ for a sequence of variables of some unspecified length $x_1, x_2, \ldots, x_m$. A term can be either a bound variable x, or a **let**-binding of y in subterm t to either a new tuple $(\overline{x})$ ($\textbf{let } y = (\overline{x}) \textbf{ in } t$), or the result of calling function f on argument x ($\textbf{let } y = f\ x \textbf{ in } t$). Both f and x are variables to be looked up in the environment. Terms cannot contain λ-abstractions as they have been λ-lifted to top-level definitions, which we encode as closures in the initial environments. Unlike standard ANF we add no special syntax for function calls in tail position (see Sec. 17.5.3 for a discussion about this limitation). We often inspect the result of a function call $f\ x$, which is not a valid term in our syntax. To enable this, we write "$@(f, x)$" for "$\textbf{let } y = f\ x \textbf{ in } y$" where y is chosen fresh.

Our syntax for change terms dt mimicks the syntax for base terms except that (i) each function call is immediately followed by the evaluation of its derivative and (ii) the final value returned by a change term is a change variable dx. As we will see, this *ad-hoc* syntax of change terms is tailored to capture the programs produced by differentiation. We only allows α-renamings that maintain the invariant that the definition of x is immediately followed by the definition of dx: if x is renamed to y then dx must be renamed to dy.

Semantics A closed value for base terms is either a closure, a tuple of values, a constants or a primitive. A closure is a pair of an evaluation environment E and a λ-abstraction closed with respect to E. The set of available constants is left abstract. It may contain usual first-order constants like integers. We also leave abstract the primitives $\mathbf{p}$ like `if-then-else` or projections of tuple components. As usual, environments E map variables to closed values. With no loss of generality, we assume that all bound variables are distinct.

Figure 17.2 shows a step-indexed big-step semantics for base terms, defined through judgment $E \vdash t \Downarrow_n v$, pronounced "Under environment E, base term t evaluates to closed value v in n steps." Our

$$\text{[SDVar]}\ \frac{}{dE \vdash dx \Downarrow dE(dx)} \qquad \text{[SDTuple]}\ \frac{dE(\overline{x}, \overline{dx}) = \overline{v}_x, \overline{dv}_x \quad dE; y = (\overline{v}_x); dy = (\overline{dv}_x) \vdash dt \Downarrow dv}{dE \vdash \textbf{let}\ y = (\overline{x}), dy = (\overline{dx})\ \textbf{in}\ dt \Downarrow dv}$$

$$\text{[SDReplaceCall]}\ \frac{dE(df) = !v_f \quad \lfloor dE \rfloor_1 \vdash @(f, x) \Downarrow_n v_y \quad \lfloor dE \rfloor_2 \vdash @(f, x) \Downarrow_m v_y' \quad dE; y = v_y; dy = !v_y' \vdash dt \Downarrow dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow dv}$$

$$\text{[SDClosureNil]}\ \frac{dE(f, df) = E_f[\lambda x.\ t_f], \textbf{nil} \quad E_f; x = dE(x) \vdash t_f \Downarrow_n v_y \quad E_f; x = \lfloor dE \rfloor_2(x) \vdash t_f \Downarrow_m v_y' \quad dE; y = v_y; dy = !v_y' \vdash dt \Downarrow dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow dv}$$

$$\text{[SDPrimitiveNil]}\ \frac{dE(f, df) = \mathbf{p}, \textbf{nil} \quad dE(x, dx) = v_x, dv_x \quad dE; y = \delta_{\mathbf{p}}(v_x); dy = \Delta_{\mathbf{p}}(v_x, dv_x) \vdash dt \Downarrow dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow dv}$$

$$\text{[SDClosureChange]}\ \frac{dE(f, df) = E_f[\lambda x.\ t_f], dE_f[\lambda x\ dx.\ dt_f] \quad dE(x, dx) = v_x, dv_x \quad E_f; x = v_x \vdash t_f \Downarrow_n v_y \quad dE_f; x = v_x; dx = dv_x \vdash dt_f \Downarrow dv_y \quad dE; y = v_y; dy = dv_y \vdash dt \Downarrow dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow dv}$$

Figure 17.3: Step-indexed big-step semantics for the change terms of the source language λ_{AL}.

step-indexes count the number of "nodes" of a big-step derivation.[1] We explain each rule in turn. Rule [SVar] looks variable x up in environment E. Other rules evaluate **let**-binding **let** $y = \ldots$ **in** t in environment E: Each rule computes y's new value v_y (taking m steps, where m can be zero) and evaluates in n steps body t to v, using environment E extended by binding y to v_y. The overall **let**-binding evaluates to v in $m + n + 1$ steps. But different rules compute y's value differently. [STuple] looks each variable in $\overline{x}$ up in E to evaluate tuple $(\overline{x})$ (in $m = 0$ steps). [SPrimitiveCall] evaluates function calls where variable f is bound in E to a primitive $\mathbf{p}$. How primitives are evaluated is specified by function $\delta_{\mathbf{p}}(-)$ from closed values to closed values. To evaluate such a primitive call, this rule applies $\delta_{\mathbf{p}}(-)$ to x's value (in $m = 0$ steps). [SClosureCall] evaluates functions calls where variable f is bound in E to closure $E_f[\lambda x.\ t_f]$: this rule evaluates closure body t_f in m steps, using closure environment E_f extended with the value of argument x in E.

Change semantics We move on to define how change terms evaluate to change values. We start by required auxiliary definitions.

A closed change value is either a closure change, a tuple change, a literal change, a replacement change or a nil change. A closure change is a pair made of an evaluation environment dE and a λ-abstraction expecting a value and a change value as arguments to evaluate a change term into an output change value. An evaluation environment dE follows the same structure as **let**-bindings of change terms: it binds variables to closed values and each variable x is immediately followed by a binding for its associated change variable dx. As with **let**-bindings of change terms, α-renamings in an environment dE must rename dx into dy if x is renamed into y. With no loss of generality, we assume that all bound term variables are distinct in these environments.

We define the *original environment* $\lfloor dE \rfloor_1$ and the *new environment* $\lfloor dE \rfloor_2$ of a change environ-

[1] Instead, Ahmed [2006] and Acar et al. [2008] count the number of steps that small-step evaluation would take (as we did in Appendix C), but this simplifies some proof steps and makes a minor difference in others.

ment dE by induction over dE:

$$\begin{array}{rcl} \lfloor \bullet \rfloor_i & = & \bullet \qquad \mathrm{i} = 1, 2 \\ \lfloor dE; x = v; dx = dv \rfloor_1 & = & \lfloor dE \rfloor_1; x = v \\ \lfloor dE; x = v; dx = dv \rfloor_2 & = & \lfloor dE \rfloor_2; x = v \oplus dv \end{array}$$

This last rule makes use of an operation $\oplus$ to update a value with a change, which may fail at runtime. Indeed, change update is a partial function written "$v \oplus dv$", defined as follows:

$$\begin{array}{rcll} v & \oplus & \mathbf{nil} & = & v \\ v_1 & \oplus & !v_2 & = & v_2 \\ \ell & \oplus & d\ell & = & \delta_\oplus(\ell, d\ell) \\ E[\lambda x.\, t] & \oplus & dE[\lambda x\, dx.\, dt] & = & (E \oplus dE)[\lambda x.\, t] \\ (v_1, \ldots, v_n) & \oplus & (dv_1, \ldots, dv_n) & = & (v_1 \oplus dv_1, \ldots, v_n \oplus dv_n) \end{array}$$

where

$$(E; x = v) \oplus (dE; x = v; dx = dv) = ((E \oplus dE); x = (v \oplus dv))$$

Nil and replacement changes can be used to update all values (constants, tuples, primitives and closures), while tuple changes can only update tuples, literal changes can only update literals and closure changes can only update closures. A nil change leaves a value unchanged. A replacement change overrides the current value v with a new one v'. On literals, $\oplus$ is defined via some interpretation function $\delta_\oplus$. Change update for a closure ignores dt instead of combining it with t somehow. This may seem surprising, but we only need $\oplus$ to behave well for valid changes (as shown by Lemma 17.3.1): for valid closure changes, dt must behave similarly to $\mathcal{D}'[\![\, t \,]\!]$ anyway, so only environment updates matter. This definition also avoids having to modify terms at runtime, which would be difficult to implement safely.

Having given these definitions, we show in Fig. 17.3 a big-step semantics for change terms (without step-indexing), defined through judgment $dE \vdash dt \Downarrow dv$, pronounced "Under the environment dE, the change term dt evaluates into the closed change value dv." [SDVar] looks up into dE to return a value for dx. [SDTuple] builds a tuple out of the values of $\overline{x}$ and a change tuple out of the change values of $\overline{dx}$ as found in the environment dE. There are four rules to evaluate **let**-binding depending on the nature of $dE(df)$. These four rules systematically recomputes the value v_y of y in the original environment. They differ in the way they compute the change dy to y.

If $dE(df)$ is a replacement, [SDReplaceCall] applies. Replacing the value of f in the environment forces recomputing $f\ x$ from scratch in the new environment. The resulting value v'_y is the new value which must replace v_y, so dy binds to $!v'_y$ when evaluating the **let** body.

If $dE(df)$ is a nil change, we have two rules depending on the nature of $dE(f)$. If $dE(f)$ is a closure, [SDClosureNil] applies and in that case the nil change of $dE(f)$ is the exact same closure. Hence, to compute dy, we reevaluate this closure applied to the updated argument $\lfloor dE \rfloor_2(x)$ to a value v'_y and bind dy to $!v'_y$. In other words, this rule is equivalent to [SDReplaceCall] in the case where a closure is replaced by itself.[2] If $dE(f)$ is a primitive, [SDPrimitiveNil] applies. The nil change of a primitive $\mathbf{p}$ is its derivative which interpretation is realized by a function $\Delta_{\mathbf{p}}(-)$. The evaluation of this function on the input value and the input change leads to the change dv_y bound to dy.

If $dE(f)$ is a closure change $dE_f[\lambda x\, dx.\, dt_f]$, [SDClosureChange] applies. The change dv_y results from the evaluation of dt_f in the closure change environment dE_f augmented with an input value for x and a change value for dx. Again, let us recall that we will maintain the invariant that the term dt_f behaves as the derivative of f so this rule can be seen as the invocation of f's derivative.

[2] Based on Appendix C, we are confident we could in fact use the derivative of t_f instead of replacement changes, but transforming terms in a semantics seems aesthetically wrong. We can also restrict **nil** to primitives, as we essentially did in Appendix C.

Expressiveness A closure in the initial environment can be used to represent a top-level definition. Since environment entries can point to primitives, we need no syntax to directly represent calls of primitives in the syntax of base terms. To encode in our syntax a program with top-level definitions and a term to be evaluated representing the entry point, one can produce a term t representing the entry point together with an environment E containing as values any top-level definitions, primitives and constants used in the program.

Our formalization does not model directly n-ary functions, but they can be encoded through unary functions and tuples. This encoding does not support currying efficiently, but we discuss possible solutions in Sec. 17.5.7.

Control operators, like recursion combinators or branching, can be introduced as primitive operations as well. If the branching condition changes, expressing the output change in general requires replacement changes. Similarly to branching we can add tagged unions.

17.3.2 Static differentiation in λ_{AL}

Definition 10.5.1 defines differentiation for simply-typed λ-calculus terms. Figure 17.4 shows differentiation for λ_{AL} syntax.

$$\begin{aligned}
\mathcal{D}^\iota[\![\, x \,]\!] &= dx \\
\mathcal{D}^\iota[\![\, \mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t \,]\!] &= \mathbf{let}\ y = (\overline{x}), dy = (\overline{dx})\ \mathbf{in}\ \mathcal{D}^\iota[\![\, t \,]\!] \\
\mathcal{D}^\iota[\![\, \mathbf{let}\ y = f\ x\ \mathbf{in}\ t \,]\!] &= \mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ \mathcal{D}^\iota[\![\, t \,]\!]
\end{aligned}$$

Figure 17.4: Static differentiation in λ_{AL}.

Differentiating a base term t produces a change term $\mathcal{D}^\iota[\![\, t \,]\!]$, its *derivative*. Differentiating final result variable x produces its change variable dx. Differentiation copies each binding of an intermediate result y to the output and adds a new bindings for its change dy. If y is bound to tuple $(\overline{x})$, then dy will be bound to the change tuple $(\overline{dx})$. If y is bound to function application $f\ x$, then dy will be bound to the application of function change df to input x and its change dx.

Evaluating $\mathcal{D}^\iota[\![\, t \,]\!]$ recomputes all intermediate results computed by t. This recomputation will be avoided through cache-transfer style in Sec. 17.3.5.

The original transformation for static differential of λ-terms [Cai et al., 2014] has three cases which we recall here:

$$\begin{aligned}
\mathrm{Derive}(x) &= dx \\
\mathrm{Derive}(t\ u) &= \mathrm{Derive}(t)\ u\ \mathrm{Derive}(u) \\
\mathrm{Derive}(\lambda x.\ t) &= \lambda x\ dx.\ \mathrm{Derive}(t)
\end{aligned}$$

Even though the first two cases of Cai et al.'s differentiation map into the two cases of our differentiation variant, one may ask where the third case is realized now. Actually, this third case occurs while we transform the initial environment. Indeed, we will assume that the closures of the environment of the source program have been adjoined a derivative. More formally, we suppose that the derivative of t is evaluated under an environment $\mathcal{D}^\iota[\![\, E \,]\!]$ obtained as follows:

$$\begin{aligned}
\mathcal{D}^\iota[\![\, \bullet \,]\!] &= \bullet \\
\mathcal{D}^\iota[\![\, E; f = E_f[\lambda x.\ t] \,]\!] &= \mathcal{D}^\iota[\![\, E \,]\!]; f = E_f[\lambda x.\ t], df = \mathcal{D}^\iota[\![\, E_f \,]\!][\lambda x\ dx.\ \mathcal{D}^\iota[\![\, t \,]\!]] \\
\mathcal{D}^\iota[\![\, E; x = v \,]\!] &= \mathcal{D}^\iota[\![\, E \,]\!]; x = v, dx = \mathbf{nil} \qquad \text{(If } v \text{ is not a closure.)}
\end{aligned}$$

17.3.3 A new soundness proof for static differentiation

As in Cai et al. [2014]'s development, static differentiation is only sound on input changes that are *valid*. However, our definition of validity needs to be significantly different. Cai et al. prove soundness for a strongly normalizing simply-typed λ-calculus using denotational semantics. We generalize this result to an untyped and Turing-complete language using purely syntactic arguments. In both scenarios, a function change is only valid if it turns valid input changes into valid output changes, so validity is a logical relation. Since standard logical relations only apply to typed languages, we turn to *step-indexed* logical relations.

Validity as a step-indexed logical relation

To state and prove soundness of differentiation, we define validity by introducing a ternary step-indexed relation over base values, changes and updated values, following previous work on step-indexed logical relations [Ahmed, 2006; Acar et al., 2008]. Experts might notice small differences in our step-indexing, as mentioned in Sec. 17.3.1, but they do not affect the substance of the proof. We write

$$dv \triangleright_k v_1 \hookrightarrow v_2$$

and say that "dv is a valid change from v_1 to v_2, up to k steps" to mean that dv is a change from v_1 to v_2 and that dv is a *valid* description of the differences between v_1 and v_2, with validity tested with up to k steps. To justify this intuition of validity, we state and prove two lemmas: a valid change from v_1 to v_2 goes indeed from v_1 to v_2 (Lemma 17.3.1), and if a change is valid up to k steps it is also valid up to fewer steps (Lemma 17.3.2).

Lemma 17.3.1 ($\oplus$ agrees with validity)
If we have $dv \triangleright_k v_1 \hookrightarrow v_2$ for all step-indexes k, then $v_1 \oplus dv = v_2$. □

Lemma 17.3.2 (Downward-closure)
If $N \geq n$, then $dv \triangleright_N v_1 \hookrightarrow v_2$ implies $dv \triangleright_n v_1 \hookrightarrow v_2$. □

- $d\ell \triangleright_n \ell \hookrightarrow \delta_\oplus(\ell, d\ell)$
- $!v_2 \triangleright_n v_1 \hookrightarrow v_2$
- $\mathbf{nil} \triangleright_n v \hookrightarrow v$
- $(dv_1, \ldots, dv_m) \triangleright_n (v_1, \ldots, v_m) \hookrightarrow (v'_1, \ldots, v'_m)$
  if and only if
  $\forall k < n, \forall i \in [1 \ldots m], dv_i \triangleright_n v_i \hookrightarrow v'_i$
- $dE[\lambda x\, dx.\, dt] \triangleright_n E_1[\lambda x.\, t] \hookrightarrow E_2[\lambda x.\, t]$
  if and only if $E_2 = E_1 \oplus dE$ and
  $\forall k < n, v_1, dv, v_2,$
  if $dv \triangleright_k v_1 \hookrightarrow v_2$
  then
  $(dE; dv \vdash dt) \blacktriangleright_k (E_1; x = v_1 \vdash t) \hookrightarrow$
  $(E_2; x = v_2 \vdash t)$

Figure 17.5: Step-indexed relation between values and changes.

As usual with step-indexed logical relations, validity is defined by well-founded induction over naturals ordered by $<$. To show this, it helps to observe that evaluation always takes at least one step.

Validity is formally defined by cases in Figure 17.5; we describe in turn each case. First, a constant change $d\ell$ is a valid change from ℓ to $\ell \oplus d\ell = \delta_\oplus(\ell, d\ell)$. Since the function $\delta_\oplus$ is partial, the relation only holds for the constant changes $d\ell$ which are valid changes for ℓ. Second, a replacement change $!v_2$ is always a valid change from any value v_1 to v_2. Third, a nil change is a valid change between any value and itself. Fourth, a tuple change is valid up to step n, if each of its components is valid up to any step strictly less than k. Fifth, we define validity for closure changes. Roughly

speaking, this statement means that a closure change is valid if (i) its environment change dE is valid for the original closure environment E_1 and for the new closure environment E_2; (ii) when applied to related values, the closure *bodies* t_1 and t_2 are related by dt. The validity relation between terms is defined as follows:

$$
\begin{array}{l}
(dE \vdash dt) \blacktriangleright_n (E_1 \vdash t_1) \hookrightarrow (E_2 \vdash t_2) \\
\text{if and only if } \forall k < n, v_1, v_2, \\
\quad E_1 \vdash t_1 \Downarrow_k v_1 \text{ and } E_2 \vdash t_2 \Downarrow v_2 \\
\quad \text{implies that } \exists dv, dE \vdash dt \Downarrow dv \wedge dv \triangleright_{n-k} v_1 \hookrightarrow v_2
\end{array}
$$

We extend this relation from values to environments by defining a judgment $dE \triangleright_k E_1 \hookrightarrow E_2$ defined as follows:

$$
\bullet \triangleright_k \bullet \hookrightarrow \bullet \qquad \frac{dE \triangleright_k E_1 \hookrightarrow E_2 \qquad dv \triangleright_k v_1 \hookrightarrow v_2}{(dE; x = v_1; dx = dv) \triangleright_k (E_1; x = v_1) \hookrightarrow E_2; x = v_2}
$$

The above lemmas about validity for values extend to environments.

Lemma 17.3.3 ($\oplus$ agrees with validity, for environments)
If $dE \triangleright_k E_1 \hookrightarrow E_2$ then $E_1 \oplus dE = E_2$. □

Lemma 17.3.4 (Downward-closure, for environments)
If $N \geq n$, then $dE \triangleright_N E_1 \hookrightarrow E_2$ implies $dE \triangleright_n E_1 \hookrightarrow E_2$. □

Finally, for both values, terms and environments, omitting the step count k from validity means validity holds for all ks. That is, for instance, $dv \triangleright v_1 \hookrightarrow v_2$ means $dv \triangleright_k v_1 \hookrightarrow v_2$ for all k.

Soundness of differentiation

We can state a soundness theorem for differentiation without mentioning step-indexes. Instead of proving it directly, we must first prove a more technical statement (Lemma 17.3.6) that mentions step-indexes explicitly.

Theorem 17.3.5 (Soundness of differentiation in λ_{AL})
If dE is a valid change environment from base environment E_1 to updated environment E_2, that is $dE \triangleright E_1 \hookrightarrow E_2$, and if t converges both in the base and updated environment, that is $E_1 \vdash t \Downarrow v_1$ and $E_2 \vdash t \Downarrow v_2$, then $\mathcal{D}^\iota[\![t]\!]$ evaluates under the change environment dE to a valid change dv between base result v_1 and updated result v_2, that is $dE \vdash \mathcal{D}^\iota[\![t]\!] \Downarrow dv$, $dv \triangleright v_1 \hookrightarrow v_2$ and $v_1 \oplus dv = v_2$. □

Lemma 17.3.6 (Fundamental Property)
For each n, if $dE \triangleright_n E_1 \hookrightarrow E_2$ then $(dE \vdash \mathcal{D}^\iota[\![t]\!]) \blacktriangleright_n (E_1 \vdash t) \hookrightarrow (E_2 \vdash t)$. □

17.3.4 The target language $i\lambda_{AL}$

In this section, we present the target language of a transformation that extends static differentiation with CTS conversion. As said earlier, the functions of $i\lambda_{AL}$ compute both their output and a cache, which contains the intermediate values that contributed to that output. Derivatives receive this cache and use it to compute changes without recomputing intermediate values; derivatives also update the caches according to their input changes.

		Base terms	
M	::=	**let** $y, c^y_{fx} = f\ x$ **in** M	*Call*
	|	**let** $y = (\overline{x})$ **in**	*Tuple*
	|	(x, C)	*Result*
		Cache terms	
C	::=	$\bullet$	*Empty*
	|	$C\ c^y_{fx}$	*Sub-cache*
	|	$C\ x$	*Cached value*
		Change terms	
dM	::=	**let** $dy, c^y_{fx} = df\ dx\ c^y_{fx}$ **in** dM	*Call*
	|	**let** $dy = (\overline{dx})$ **in** dM	*Tuple*
	|	(dx, dC)	*Result*
		Cache updates	
dC	::=	$\bullet$	*Empty*
	|	$dC\ c^y_{fx}$	*Sub-cache*
	|	$dC\ (x \oplus dx)$	*Updated value*
		Closed values	
V	::=	$F[\lambda x.\ M]$	*Closure*
	|	$(\overline{V})$	*Tuple*
	|	ℓ	*Literal*
	|	**p**	*Primitive*

		Cache values	
V_c	::=	$\bullet$	*Empty*
	|	$V_c\ V_c$	*Sub-cache*
	|	$V_c\ V$	*Cached value*
		Change values	
dV	::=	$dF[\lambda dx\, C.\ dM]$	*Closure*
	|	$(\overline{dV})$	*Tuple*
	|	$d\ell$	*Literal*
	|	**nil**	*Nil*
	|	$!V$	*Replacement*
		Base definitions	
D_v	::=	$x = V$	*Value definition*
	|	$c^y_{fx} = V_c$	*Cache definition*
		Change definitions	
dD_v	::=	D_v	*Base*
	|	$dx = dV$	*Change*
		Evaluation environments	
F	::=	$F; D_v$	*Binding*
	|	$\bullet$	*Empty*
		Change environments	
dF	::=	$dF; dD_v$	*Binding*
	|	$\bullet$	*Empty*

Figure 17.6: Target language $i\lambda_{AL}$ (syntax).

Syntax The syntax of $i\lambda_{AL}$ is defined in Figure 17.6. Base terms of $i\lambda_{AL}$ follow again λ-lifted A'NF, like λ_{AL}, except that a **let**-binding for a function application $f\ x$ now binds an extra *cache identifier* c^y_{fx} besides output y. Cache identifiers have non-standard syntax: it can be seen as a triple that refers to the value identifiers f, x and y. Hence, an α-renaming of one of these three identifiers must refresh the cache identifier accordingly. Result terms explicitly return cache C through syntax (x, C).

The syntax for caches has three cases: a cache can be empty, or it can prepend a value or a cache variable to a cache. In other words, a cache is a tree-like data structure which is isomorphic to an execution trace containing both immediate values and the execution traces of the function calls issued during the evaluation.

The syntax for change terms of $i\lambda_{AL}$ witnesses the CTS discipline followed by the derivatives: to determine dy, the derivative of f evaluated at point x with change dx expects the cache produced by evaluating y in the base term. The derivative returns the updated cache which contains the intermediate results that would be gathered by the evaluation of $f\ (x \oplus dx)$. The result term of every change term returns a cache update dC in addition to the computed change.

The syntax for cache updates resembles the one for caches, but each value identifier x of the input cache is updated with its corresponding change dx.

Semantics An evaluation environment F of $i\lambda_{AL}$ contains not only values but also cache values. The syntax for values V includes closures, tuples, primitives and constants. The syntax for cache values V_c mimics the one for cache terms. The evaluation of change terms expects the evaluation environments dF to also include bindings for change values.

There are five kinds of change values: closure changes, tuple changes, literal changes, nil changes and replacements. Closure changes embed an environment dF and a code pointer for a function, waiting for both a base value x and a cache C. By abuse of notation, we reuse the same syntax C to both deconstruct and construct caches. Other changes are similar to the ones found in λ_{AL}.

Base terms of the language are evaluated using a big-step semantics defined in Figure 17.7.

Evaluation of base terms $\boxed{F \vdash M \Downarrow (V, V_c)}$

$$\text{[TResult]}\ \frac{F(x) = V \qquad F \vdash C \Downarrow V_c}{F \vdash (x, C) \Downarrow (V, V_c)}$$

$$\text{[TTuple]}\ \frac{F; y = F(\overline{x}) \vdash M \Downarrow (V, V_c)}{F \vdash \textbf{let}\ y = (\overline{x})\ \textbf{in}\ M \Downarrow (V, V_c)}$$

$$\text{[TClosureCall]}\ \frac{\begin{array}{c} F(f) = F'[\lambda x'.\, M'] \\ F'; x' = F(x) \vdash M' \Downarrow (V', V_c') \\ F; y = V'; c_{fx}^{y} = V_c' \vdash M \Downarrow (V, V_c) \end{array}}{F \vdash \textbf{let}\ y, c_{fx}^{y} = f\ x\ \textbf{in}\ M \Downarrow (V, V_c)}$$

$$\text{[TPrimitiveCall]}\ \frac{F(f) = \ell \qquad \delta_\ell(F(x)) = (V', V_c') \qquad F; y = V'; c_{fx}^{y} = V_c' \vdash M \Downarrow v}{F \vdash \textbf{let}\ y, c_{fx}^{y} = f\ x\ \textbf{in}\ M \Downarrow (V, V_c)}$$

Evaluation of caches $\boxed{F \vdash C \Downarrow V_c}$

$$\text{[TEmptyCache]}\ \frac{}{F \vdash \bullet \Downarrow \bullet}$$

$$\text{[TCacheVar]}\ \frac{F(x) = V \qquad F \vdash C \Downarrow V_c}{F \vdash C\,x \Downarrow V_c\,V}$$

$$\text{[TCacheSubCache]}\ \frac{F(c_{fx}^{y}) = V_c' \qquad F \vdash C \Downarrow V_c}{F \vdash C\,c_{fx}^{y} \Downarrow V_c\,V_c'}$$

Figure 17.7: Target language $i\lambda_{AL}$ (semantics of base terms and caches).

Judgment "$F \vdash M \Downarrow (V, V_c)$" is read "Under evaluation environment F, base term M evaluates to value V and cache V_c". Auxiliary judgment "$F \vdash C \Downarrow V_c$" evaluates cache terms into cache values. Rule [TResult] not only looks into the environment for the return value V but it also evaluates the returned cache C. Rule [TTuple] is similar to the rule of the source language since no cache is produced by the allocation of a tuple. Rule [TClosureCall] works exactly as [SClosureCall] except that the cache value returned by the closure is bound to cache identifier c_{fx}^{y}. In the same way, [TPrimitiveCall] resembles [SPrimitiveCall] but also binds c_{fx}^{y}. Rule [TEmptyCache] evaluates an empty cache term into an empty cache value. Rule [TCacheVar] computes the value of cache term $C\,x$ by appending the value of variable x to cache value V_c computed for cache term C. Similarly, rule [TCacheSubCache] appends the cache value of a cache named c_{fx}^{y} to the cache value V_c computed for C.

Change terms of the target language are also evaluated using a big-step semantics, defined in Fig. 17.8. Judgment "$dF \vdash dM \Downarrow (dV, V_c)$" is read "Under evaluation environment F, change term dM evaluates to change value dV and updated cache V_c". The first auxiliary judgment "$dF \vdash dC \Downarrow V_c$" defines evaluation of cache update terms. We omit the rules for this judgment since it is similar to the one for cache terms, except that cached values are computed by $x \oplus dx$, not simply x. The final auxiliary judgment "$V_c \sim C \longrightarrow dF$" describes a limited form of pattern matching used by CTS derivatives: namely, how a cache pattern C matches a cache value V_c to produce a change environment dF.

Rule [TDResult] returns the final change value of a computation as well as a updated cache resulting from the evaluation of the cache update term dC. Rule [TDTuple] resembles its counterpart in the source language, but the tuple for y is not built as it has already been pushed in the environment by the cache.

As for λ_{AL}, there are four rules to deal with **let**-bindings depending on the shape of the change bound to df in the environment. If df is bound to a replacement, the rule [TDReplaceCall] applies.

Evaluation of change terms $\boxed{dF \vdash dM \Downarrow (dV, V_c)}$

[TDResult]
$$\frac{dF(dx) = dV \qquad dF \vdash dC \Downarrow V_c}{dF \vdash (dx, dC) \Downarrow (dV, V_c)}$$

[TDTuple]
$$\frac{dF; dy = dF(\overline{dx}) \vdash dM \Downarrow (dV, V_c)}{dF \vdash \textbf{let } dy = (\overline{dx}) \textbf{ in } dM(dV, V_c) \Downarrow}$$

[TDReplaceCall]
$$\frac{\begin{array}{c} dF(df) = !V_f \\ \lfloor dF \rfloor_2 \vdash @(f, x) \Downarrow (V', V_c') \\ dF; dy = !V'; c_{fx}^y = V_c' \vdash dM \Downarrow (dV, V_c) \end{array}}{dF \vdash \textbf{let } dy, c_{fx}^y = df\ dx\ c_{fx}^y \textbf{ in } dM \Downarrow (dV, V_c)}$$

[TDClosureNil]
$$\frac{\begin{array}{c} dF(f, df) = dF_f[\lambda x.\, M_f], \textbf{nil} \\ dF_f; x = \lfloor dF \rfloor_2(x) \vdash M_f \Downarrow (V_y', V_c') \\ dF; dy = !V_y'; c_{fx}^y = V_c' \vdash dM \Downarrow (dV, V_c) \end{array}}{dF \vdash \textbf{let } dy, c_{fx}^y = df\ dx\ c_{fx}^y \textbf{ in } dM \Downarrow (dV, V_c)}$$

[TDPrimitiveNil]
$$\frac{dF(f, df) = \mathbf{p}, \textbf{nil} \qquad dF(x, dx) = V_x, dV_x \qquad dE; dy, c_{fx}^y = \Delta_{\mathbf{p}}(V_x, dV_x, dF(c_{fx}^y)) \vdash dM \Downarrow (dV, V_c)}{dF \vdash \textbf{let } dy, c_{fx}^y = df\ dx\ c_{fx}^y \textbf{ in } dM \Downarrow (dV, V_c)}$$

[TDClosureChange]
$$\frac{\begin{array}{c} dF(df) = dF_f[\lambda dx\, C.\, dM_f] \qquad dF(c_{fx}^y) \sim C \to dF' \\ dF_f; dx = dF(dx); dF' \vdash dM_f \Downarrow (dV_y, V_c') \qquad dF; dy = dV_y, c_{fx}^y = V_c' \vdash dM \Downarrow (dV, V_c) \end{array}}{dF \vdash \textbf{let } dy, c_{fx}^y = df\ dx\ c_{fx}^y \textbf{ in } dM \Downarrow (dV, V_c)}$$

Binding of caches $\boxed{V_c \sim C \to dF}$

[TMatchEmptyCache]
$$\frac{}{\bullet \sim \bullet \to \bullet}$$

[TMatchCachedValue]
$$\frac{V_c \sim C \to dF}{V_c\ V \sim C\ x \to dF; (x = V)}$$

[TMatchSubCache]
$$\frac{V_c \sim C \to dF}{V_c\ V_c' \sim C\ c_{fx}^y \to dF; (c_{fx}^y = V_c')}$$

Figure 17.8: Target language $i\lambda_{AL}$ (semantics of change terms and cache updates).

In that case, we reevaluate the function call in the updated environment $\lfloor dF \rfloor_2$ (defined similarly as in the source language). This evaluation leads to a new value V' which replaces the original one as well as an updated cache for c_{fx}^y.

If df is bound to a nil change and f is bound to a closure, the rule [TDClosureNil] applies. This rule mimicks again its counterpart in the source language passing with the difference that only the resulting change and the updated cache are bound in the environment.

If df is bound to a nil change and f is bound to primitive p, the rule [TDPrimitiveNil] applies. The derivative of $\mathbf{p}$ is invoked with the value of x, its change value and the cache of the original call to $\mathbf{p}$. The semantics of $\mathbf{p}$'s derivative is given by builtin function $\Delta_{\mathbf{p}}(-)$, as in the source language.

If df is bound to a closure change and f is bound to a closure, the rule [TDClosureChange] applies. The body of the closure change is evaluated under the closure change environment extended with the value of the formal argument dx and with the environment resulting from the binding of the original cache value to the variables occuring in the cache C. This evaluation leads to a change and an updated cache bound in the environment to continue with the evaluation of the rest of the term.

CTS translation of toplevel definitions $\boxed{\mathcal{C}[\![\, f = \lambda x.\, t \,]\!]}$

$$\begin{aligned}
\mathcal{C}[\![\, f = \lambda x.\, t \,]\!] &= f = \lambda x.\, M,\\
&\quad\ df = \lambda dx\, C.\, \mathcal{D}_{\bullet\,(x \oplus dx)}[\![\, t \,]\!]\\
&\quad \text{where } (C, M) = \mathcal{C}_{\bullet\, x}[\![\, t \,]\!]
\end{aligned}$$

CTS differentiation of terms $\boxed{\mathcal{D}_{dC}[\![\, t \,]\!]}$

$$\begin{aligned}
\mathcal{D}_{dC}[\![\, \textbf{let } y = f\ x \textbf{ in } t \,]\!] &= [\![\, \textbf{let } dy, c^y_{fx} = df\ dx\ c^y_{fx} \textbf{ in } M \,]\!]\\
&\quad \text{where } M = \mathcal{D}_{(dC\ (y \oplus dy)\ c^y_{fx})}[\![\, t \,]\!]\\
\mathcal{D}_{dC}[\![\, \textbf{let } y = (\overline{x}) \textbf{ in } t \,]\!] &= [\![\, \textbf{let } dy = (\overline{dx}) \textbf{ in } M \,]\!]\\
&\quad \text{where } M = \mathcal{D}_{(dC\ (y \oplus dy))}[\![\, t \,]\!]\\
\mathcal{D}_{dC}[\![\, x \,]\!] &= [\![\, (dx, dC) \,]\!]
\end{aligned}$$

CTS translation of terms $\boxed{\mathcal{C}_C[\![\, t \,]\!]}$

$$\begin{aligned}
\mathcal{C}_C[\![\, \textbf{let } y = f\ x \textbf{ in } t \,]\!] &= (C', [\![\, \textbf{let } y, c^y_{fx} = f\ x \textbf{ in } M \,]\!])\\
&\quad \text{where } (C', M) = \mathcal{C}_{(C\ y\ c^y_{fx})}[\![\, t \,]\!]\\
\mathcal{C}_C[\![\, \textbf{let } y = (\overline{x}) \textbf{ in } t \,]\!] &= (C', [\![\, \textbf{let } y = (\overline{x}) \textbf{ in } M \,]\!])\\
&\quad \text{where } (C', M) = \mathcal{C}_{(C\ y)}[\![\, t \,]\!]\\
\mathcal{C}_C[\![\, x \,]\!] &= (C, [\![\, (x, C) \,]\!])
\end{aligned}$$

Figure 17.9: Cache-Transfer Style (CTS) conversion from λ_{AL} to $i\lambda_{AL}$.

17.3.5 CTS conversion from λ_{AL} to $i\lambda_{AL}$

CTS conversion from λ_{AL} to $i\lambda_{AL}$ is defined in Figure 17.9. It comprises CTS differentiation $\mathcal{D}[\![-]\!]$, from λ_{AL} base terms to $i\lambda_{AL}$ change terms, and CTS translation $\mathcal{C}[\![-]\!]$, from λ_{AL} to $i\lambda_{AL}$, which is overloaded over top-level definitions $\mathcal{C}[\![\, f = \lambda x.\, t \,]\!]$ and terms $\mathcal{C}_C[\![\, t \,]\!]$.

By the first rule, $\mathcal{C}[\![-]\!]$ maps each source toplevel definition "$f = \lambda x.\, t$" to the compiled code of the function f and to the derivative df of f expressed in the target language $i\lambda_{AL}$. These target definitions are generated by a first call to the compilation function $\mathcal{C}_{\bullet\, x}[\![\, t \,]\!]$: it returns both M, the compiled body of f and the cache term C which contains the names of the intermediate values computed by the evaluation of M. This cache term C is used as a cache pattern to define the second argument of the derivative of f. That way, we make sure that the shape of the cache expected by df is consistent with the shape of the cache produced by f. Derivative body df is computed by derivation call $\mathcal{D}_{\bullet\,(x \oplus dx)}[\![\, t \,]\!]$.

CTS translation on terms, $\mathcal{C}_C[\![\, t \,]\!]$, accepts a term t and a cache term C. This cache is a fragment of output code: in tail position ($t = x$), it generates code to return both the result x and the cache C. When the transformation visits **let**-bindings, it outputs extra bindings for caches c^y_{fx}, and appends all variables newly bound in the output to the cache used when visiting the **let**-body.

Similarly to $\mathcal{C}_C[\![\, t \,]\!]$, CTS derivation $\mathcal{D}_{dC}[\![\, t \,]\!]$ accepts a cache update dC to return in tail position. While cache terms record *intermediate results*, cache updates record *result updates*. For **let**-bindings, to update y by change dy, CTS derivation appends to dC term $y \oplus dy$ to replace y.

CTS translation of values $\boxed{\mathcal{C}[\![\, v \,]\!]}$

$$\begin{aligned}\mathcal{C}[\![\, E[\lambda x.\, t] \,]\!] &= \mathcal{C}[\![\, E \,]\!][\lambda x.\, M] \\ &\quad \text{where } (C, M) = \mathcal{C}_{\bullet\, x}[\![\, t \,]\!] \\ \mathcal{C}[\![\, \ell \,]\!] &= \ell\end{aligned}$$

CTS translation of change values $\boxed{\mathcal{C}[\![\, dv \,]\!]}$

$$\begin{aligned}\mathcal{C}[\![\, dE[\lambda x\, dx.\, \mathcal{D}^{\iota}[\![\, t \,]\!]] \,]\!] &= \mathcal{C}[\![\, dE \,]\!][\lambda dx\, C.\, \mathcal{D}_{\bullet\, (x \oplus dx)}[\![\, t \,]\!]] \\ &\quad \text{where } (C, M) = \mathcal{C}_{\bullet\, x}[\![\, t \,]\!] \\ \mathcal{C}[\![\, !v \,]\!] &= !\mathcal{C}[\![\, v \,]\!] \\ \mathcal{C}[\![\, d\ell \,]\!] &= d\ell\end{aligned}$$

CTS translation of value environments $\boxed{\mathcal{C}[\![\, E \,]\!]}$

$$\begin{aligned}\mathcal{C}[\![\, \bullet \,]\!] &= \bullet \\ \mathcal{C}[\![\, E; x = v \,]\!] &= \mathcal{C}[\![\, E \,]\!]; x = \mathcal{C}[\![\, v \,]\!]\end{aligned}$$

CTS translation of change environments $\boxed{\mathcal{C}[\![\, dE \,]\!]}$

$$\begin{aligned}\mathcal{C}[\![\, \bullet \,]\!] &= \bullet \\ \mathcal{C}[\![\, dE; x = v, dx = dv \,]\!] &= \mathcal{C}[\![\, dE \,]\!]; x = \mathcal{C}[\![\, v \,]\!], dx = \mathcal{C}[\![\, dv \,]\!]\end{aligned}$$

Figure 17.10: Extending CTS translation to values, change values, environments and change environments.

17.3.6 Soundness of CTS conversion

In this section, we outline definitions and main lemmas needed to prove CTS conversion sound. The proof is based on a mostly straightforward simulation in lock-step, but two subtle points emerge. First, we must relate λ_{AL} environments that do not contain caches, with $i\lambda_{AL}$ environments that do. Second, while evaluating CTS derivatives, the evaluation environment mixes caches from the base computation and updated caches computed by the derivatives.

Evaluation commutes with CTS conversion Figure 17.10 extends CTS translation to values, change values, environments and change environments. CTS translation of base terms commutes with our semantics:

Lemma 17.3.7 (Evaluation commutes with CTS translation)
For all E, t and v, such that $E \vdash t \Downarrow v$, and for all C, there exists V_c, $\mathcal{C}[\![\, E \,]\!] \vdash \mathcal{C}_C[\![\, t \,]\!] \Downarrow (\mathcal{C}[\![\, v \,]\!], V_c)$.□

Stating a corresponding lemma for CTS translation of derived terms is trickier. If the derivative of t evaluates correctly in some environment (that is $dE \vdash \mathcal{D}^{\iota}[\![\, t \,]\!] \Downarrow dv$), CTS derivative $\mathcal{D}_{dC}[\![\, t \,]\!]$ cannot be evaluated in environment $\mathcal{C}[\![\, dE \,]\!]$. A CTS derivative can only evaluate against environments containing cache values from the base computation, but no cache values appear in $\mathcal{C}[\![\, dE \,]\!]$!

As a fix, we enrich $\mathcal{C}[\![\, dE \,]\!]$ with the values of a cache C, using the pair of judgments $dF \uparrow C \rightsquigarrow_i dF'$ (for $i = 1, 2$) defined in Fig. 17.11. Judgment $dF \uparrow C \rightsquigarrow_i dF'$ (for $i = 1, 2$) is read "Target change environment dF' extends dF with the original (for $i = 1$) (or updated, for $i = 2$) values of cache C." Since C is essentially a list of variables containing no values, cache values in dF' must be computed from $\lfloor dF \rfloor_i$.

$$dF \uparrow \bullet \rightsquigarrow_i dF \qquad \frac{dF \uparrow C \rightsquigarrow_i dF'}{dF \uparrow C\ x \rightsquigarrow_i dF'} \qquad \frac{dF \uparrow C \rightsquigarrow_i dF' \qquad \lfloor dF \rfloor_i \vdash \textbf{let}\ y, c^y_{fx} = f\ x\ \textbf{in}\ (y, c^y_{fx}) \Downarrow (\_, V_c)}{dF \uparrow C\ c^y_{fx} \rightsquigarrow_i dF'; c^y_{fx} = V_c}$$

Figure 17.11: Extension of an environment with cache values $dF \uparrow C \rightsquigarrow_i dF'$ (for $i = 1, 2$).

Lemma 17.3.8 (Evaluation commutes with CTS differentiation)
Let C be such that $(C, \_) = \mathcal{C}_\bullet[\![\ t\ ]\!]$. For all dE, t and dv, if $dE \vdash \mathcal{D}^\iota[\![\ t\ ]\!] \Downarrow dv$, and $\mathcal{C}[\![\ dE\ ]\!] \uparrow C \rightsquigarrow_1 dF$, then $dF \vdash \mathcal{D}_{dC}[\![\ t\ ]\!] \Downarrow (\mathcal{C}[\![\ dv\ ]\!], V_c)$. □

The proof of this lemma is not immediate, since during the evaluation of $\mathcal{D}_{dC}[\![\ t\ ]\!]$ the new caches replace the old caches. In our Coq development, we enforce a physical separation between the part of the environment containing old caches and the one containing new caches, and we maintain the invariant that the second part of the environment corresponds to the remaining part of the term.

Soundness of CTS conversion Finally, we can state soundness of CTS differentiation relative to differentiation. The theorem says that (a) the CTS derivative $\mathcal{D}_C[\![\ t\ ]\!]$ computes the CTS translation $\mathcal{C}[\![\ dv\ ]\!]$ of the change computed by the standard derivative $\mathcal{D}^\iota[\![\ t\ ]\!]$; (b) the updated cache V_{c2} produced by the CTS derivative coincides with the cache produced by the CTS-translated base term M in the updated environment $\lfloor dF_2 \rfloor_2$. We must use $\lfloor dF_2 \rfloor_2$ instead of dF_2 to evaluate CTS-translated base term M since dF_2, produced by environment extension, contains updated caches, changes and original values. Since we require a correct cache via condition (b), we can use this cache to invoke the CTS derivative on further changes, as described in Sec. 17.2.2.

Theorem 17.3.9 (Soundness of CTS differentiation wrt differentiation)
Let C and M be such that $(C, M) = \mathcal{C}_\bullet[\![\ t\ ]\!]$. For all dE, t and dv such that $dE \vdash \mathcal{D}^\iota[\![\ t\ ]\!] \Downarrow dv$ and for all dF_1 and dF_2 such that

$$\begin{aligned} \mathcal{C}[\![\ dE\ ]\!] \uparrow C &\rightsquigarrow_1 dF_1 \\ \mathcal{C}[\![\ dE\ ]\!] \uparrow C &\rightsquigarrow_2 dF_2 \\ \lfloor dF_1 \rfloor_1 \vdash M &\Downarrow (V_1, V_{c1}) \\ \lfloor dF_2 \rfloor_2 \vdash M &\Downarrow (V_2, V_{c2}) \end{aligned}$$

we have

$$dF_1 \vdash \mathcal{D}_C[\![\ t\ ]\!] \Downarrow (\mathcal{C}[\![\ dv\ ]\!], V_{c2}).$$

□

17.4 Incrementalization case studies

In this section, we investigate whether our transformations incrementalize efficiently programs in a typed language such as Haskell. And indeed, after providing support for the needed bulk operations on sequences, bags and maps, we successfully transform a few case studies and obtain efficient incremental programs, that is ones for which incremental computation is faster than from scratch recomputation.

We type CTS programs by by associating to each function f a cache type FC. We can transform programs that use higher-order functions by making closure construction explicit.

We illustrate those points on three case studies: average computation over bags of integers, a nested loop over two sequences and a more involved example inspired by Koch et al.'s work on incrementalizing database queries.

In all cases, we confirm that results are consistent between from scratch recomputation and incremental evaluation.

Our benchmarks were compiled by GHC 8.0.2. They were run on a 2.60GHz dual core Intel i7-6600U CPU with 12GB of RAM running Ubuntu 16.04.

17.4.1 Averaging integers bags

Section Sec. 17.2.2 motivates our transformation with a running example of computing the average over a bag of integers. We represent bags as maps from elements to (possibly negative) multiplicities. Earlier work [Cai et al., 2014; Koch et al., 2014] represents bag changes as bags of removed and added elements: to update element a with change da, one removes a and adds $a \oplus da$. Instead, our bag changes contain element changes: a valid bag change is either a replacement change (with a new bag) or a list of atomic changes. An atomic change contains an element a, a valid change for that element da, a multiplicity n and a change to that multiplicity dn. Updating a bag with an atomic change means removing n occurrences of a and inserting $n \oplus dn$ occurrences of updated element $a \oplus da$.

$$\begin{aligned}
&\textbf{type } \mathit{Bag}\ a = \mathit{Map}\ a\ \mathbb{Z}\\
&\textbf{type } \Delta(\mathit{Bag}\ a) = \mathit{Replace}\ (\mathit{Bag}\ a) \mid \mathit{Ch}\ [\mathit{AtomicChange}\ a]\\
&\textbf{data } \mathit{AtomicChange}\ a = \mathit{AtomicChange}\ a\ (\Delta a)\ \mathbb{Z}\ (\Delta\mathbb{Z})
\end{aligned}$$

This change type enables both users and derivatives to express efficiently various bag changes, such as inserting a single element or changing a single element, but also changing some but not all occurrences of an element in a bag.

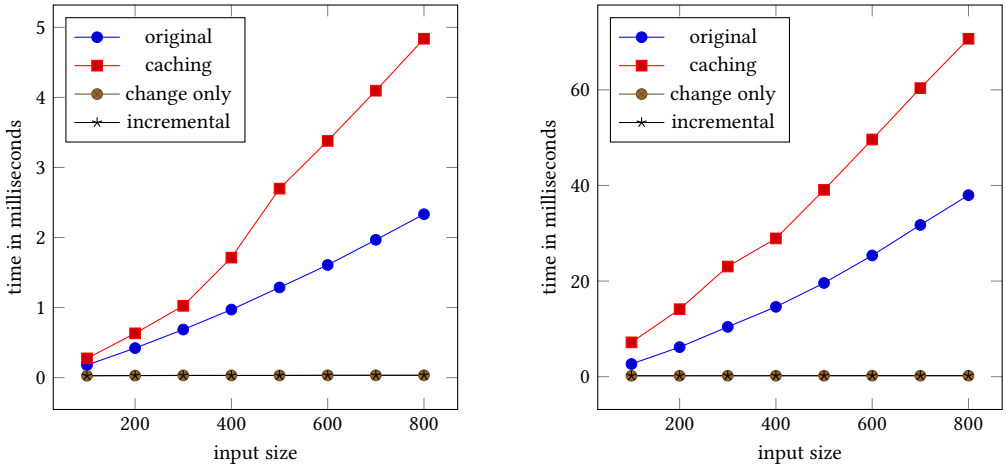
$$\begin{aligned}
&\mathit{insertSingle} :: a \to \Delta(\mathit{Bag}\ a)\\
&\mathit{insertSingle}\ a = \mathit{Ch}\ [\mathit{AtomicChange}\ a\ \mathbf{0}_a\ 0\ (\mathit{Ch}\ 1)]\\
&\mathit{changeSingle} :: a \to \Delta a \to \Delta(\mathit{Bag}\ a)\\
&\mathit{changeSingle}\ a\ da = \mathit{Ch}\ [\mathit{AtomicChange}\ a\ da\ 1\ \mathbf{0}_1]
\end{aligned}$$

Based on this change structure, we provide efficient primitives and their derivatives. The CTS variant of *map*, that we call *mapC* takes a function *fC* in CTS and a bag *as* and produces a bag and a cache. Because *map* is not self-maintainable the cache stores the arguments *fC* and *as*. In addition the cache stores for each invocation of *fC*, and therefore for each distinct element in *as*, the result of *fC* of type *b* and the cache of type *c*. The incremental function *dmapC* has to produce an updated cache. We check that this is the same cache that *mapC* would produce when applied to updated inputs using the QuickCheck library.

Following ideas inspired by Rossberg et al. [2010], all higher-order functions (and typically, also their caches) are parametric over cache types of their function arguments. Here, functions *mapC* and *dmapC* and cache type *MapC* are parametric over the cache type *c* of *fC* and *dfC*.

$$\begin{aligned}
&\mathit{map} :: (a \to b) \to \mathit{Bag}\ a \to \mathit{Bag}\ b\\
&\textbf{type } \mathit{MapC}\ a\ b\ c = (a \to (b, c), \mathit{Bag}\ a, \mathit{Map}\ a\ (b, c))\\
&\mathit{mapC} :: (a \to (b, c)) \to \mathit{Bag}\ a \to (\mathit{Bag}\ b, \mathit{MapC}\ a\ b\ c)\\
&\mathit{dmapC} :: (\Delta a \to c \to (\Delta b, c)) \to \Delta(\mathit{Bag}\ a) \to\\
&\quad \mathit{MapC}\ a\ b\ c \to (\Delta(\mathit{Bag}\ b), \mathit{MapC}\ a\ b\ c)
\end{aligned}$$

We wrote the *length* and *sum* function used in our benchmarks in terms of primitives *map* and *foldGroup*. We applied our transformations to get *lengthC*, *dlengthC*, *sumC* and *dsumC*. This transformation could in principle be automated. Implementing the CTS variant and CTS derivative of primitives efficiently requires manual work.



(a) Benchmark results for *avg*

(b) Benchmark results for *totalPrice*

We evaluate whether we can produce an updated result with *davgC* faster than by from scratch recomputation with *avg*. We use the *criterion* library for benchmarking. The benchmarking code and our raw results are available in the supplementarty material. We fix an initial bag of size n. It contains the integers from 1 to n. We define a sequence of consecutive changes of length r as the insertion of 1, the deletion of 1, the insertion of 2, the deletion of 2 and so on. We update the initial bag with these changes, one after another, to obtain a sequence of input bags.

We measure the time taken by from scratch recomputation. We apply *avg* to each input bag in the sequence and fully force all results. We report the time from scratch recomputation takes as this measured time divided by the number r of input bags. We measure the time taken by our CTS variant *avgC* similarly. We make sure to fully force all results and caches *avgC* produces on the sequence of input bags.

We measure the time taken by our CTS derivative *davgC*. We assume a fully forced initial result and cache. We apply *davgC* to each change in the sequence of bag changes using the cache from the previous invocation. We then apply the result change to the previous result. We fully force the obtained sequence of results. Finally we measure the time taken to only produce the sequence of result changes without applying them.

Because of laziness, choosing what exactly to measure is tricky. We ensure that reported times include the entire cost of computing the whole sequence of results. To measure this cost, we ensure any thunks within the sequence are forced. We need not force any partial results, such as output changes or caches. Doing so would distort the results because forcing the whole cache can cost asymptotically more than running derivatives. However, not using the cache at all underestimates the cost of incremental computation. Hence, we measure a sequence of consecutive updates, each using the cache produced by the previous one.

The plot in Fig. 17.12a shows execution time versus the size n of the initial input. To produce the initial result and cache, *avgC* takes longer than the original *avg* function takes to produce just the result. This is to be expected. Producing the result incrementally is much faster than from scratch recomputation. This speedup increases with the size of the initial bag. For an initial bag size of 800 incrementally updating the result is 70 times faster than from scratch recomputation.

The difference between only producing the output change versus producing the output change and updating the result is very small in this example. This is to be expected because in this example the result is an integer and updating and evaluating an integer is fast. We will see an example where

there is a bigger difference between the two.

17.4.2 Nested loops over two sequences

We implemented incremental sequences and related primitives following Firsov and Jeltsch [2016]: our change operations and first-order operations (such as *concat*) reuse their implementation. On the other hand, we must extend higher-order operations such as *map* to handle non-nil function changes and caching. A correct and efficient CTS incremental *dmapC* function has to work differently depending on whether the given function change is nil or not: For a non-nil function change it has to go over the input sequence; for a nil function change it can avoid that.

Consider the running example again, this time in A'NF. The partial application of a lambda lifted function constructs a closure. We made that explicit with a *closure* function.

```
cartesianProduct xs ys = let
  mapPairYs = closure mapPair ys
  xys = concatMap mapPairYs xs
  in xys
mapPair ys = λx → let
  pairX = closure pair x
  xys = map pairX ys
  in xys
```

While the only valid change for closed functions is their nil change, for closures we can have non-nil function changes. We represent closed functions and closures as variants of the same type. Correspondingly we represent changes to a closed function and changes to a closure as variants of the same type of function changes. We inspect this representation at runtime to find out if a function change is a nil change.

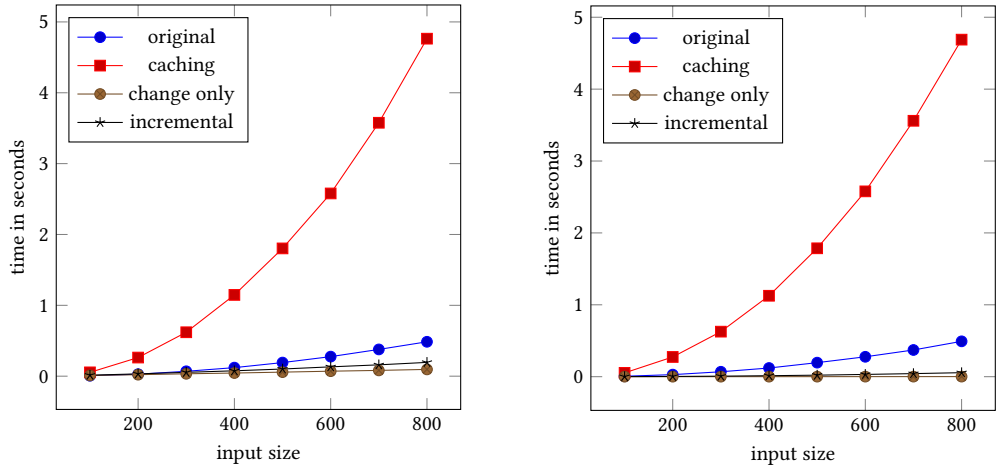
```
data Fun a b c where
  Closed :: (a → (b, c)) → Fun a b c
  Closure :: (e → a → (b, c)) → e → Fun a b c
data Δ(Fun a b c) where
  DClosed :: (Δa → c → (Δb, c)) → Δ(Fun a b c)
  DClosure :: (Δe → Δa → c → (Δb, c)) → Δe → Δ(Fun a b c)
```

We have also evaluated another representation of function changes with different tradeoffs where we use defunctionalization instead of closure conversion. We discuss the use of defunctionalization in Appendix D.

We use the same benchmark setup as in the benchmark for the average computation on bags. The input for size n is a pair of sequences (xs, ys). Each sequence initially contains the integers from 1 to n. Updating the result in reaction to a change dxs to xs takes less time than updating the result in reaction to a change dys to ys. The reason is that when dys is a nil change we dynamically detect that the function change passed to *dconcatMapc* is the nil change and avoid looping over xs entirely.

We benchmark changes to the outer sequence xs and the inner sequence ys separately. In both cases the sequence of changes are insertions and deletions of different numbers at different positions. When we benchmark changes to the outer sequence xs we take this sequence of changes for xs and all changes to ys will be nil changes. When we benchmark changes to the inner sequence ys we take this sequence of changes for ys and all changes to xs will be nil changes.

In this example preparing the cache takes much longer than from scratch recomputation. The speedup of incremental computation over from scratch recomputation increases with the size of



(a) Benchmark results for cartesian product changing *inner* sequence.

(b) Benchmark results for Cartesian product changing *outer* sequence.

the initial sequences. It reaches 2.5 for a change to the inner sequences and 8.8 for a change to the outer sequence when the initial sequences have size 800. Producing and fully forcing only the changes to the results is 5 times faster for a change to the inner sequence and 370 times faster for a change to the outer sequence.

While we do get speedups for both kinds of changes (to the inner and to the outer sequence) the speedups for changes to the outer sequence are bigger. Due to our benchmark methodology we have to fully force updated results. This alone takes time quadratic in the size of the input sequences n. Threrefore we also report the time it takes to produce and fully force only the output changes which is of a lower time complexity.

17.4.3 Indexed joins of two bags

As expected, we found that we can compose functions into larger and more complex programs and apply CTS differentiation to get a fast incremental program.

For this example imagine we have a bag of orders and a bag of line items. An order is a pair of an integer key and an exchange rate represented as an integer. A line item is a pair of a key of a corresponding order and a price represented as an integer. The price of an order is the sum of the prices of the corresponding line items multiplied by the exchange rate. We want to find the total price defined as the sum of the prices of all orders.

To do so efficiently we build an index for line items and an index for orders. We represent indexes as maps. We build a map from order key to the sum of the prices of corresponding line items. Because orders are not necessarily unique by key and because multiplication distributes over addition we can build an index mapping each order key to the sum of all exchange rates for this order key. We then compute the total price by merging the two maps by key, multiplying corresponding sums of exchange rates and sums of prices. We compute the total price as the sum of those products.

Because our indexes are maps we need a change structure for maps. We generalize the change structure for bags by generalizing from multiplicities to arbitrary values as long as they have a group structure. A change to a map is either a replacement change (with a new map) or a list of atomic changes. An atomic change is a key k, a valid change to that key dk, a value a and a valid

change to that value *da*. To update a map with an atomic change means to use the group structure of the type of *a* to subtract *a* at key *k* and to add $a \oplus da$ at key $k \oplus dk$.

```
type Δ(Map k a) = Replace (Map k a) | Ch [AtomicChange k a]
data AtomicChange k a = AtomicChange k (Δk) a (Δa)
```

Bags have a group structure as long as the elements have a group structure. Maps have a group structure as long as the values have a group structure. This means for example that we can efficiently incrementalize programs on maps from keys to bags of elements. We implemented efficient caching and incremental primitives for maps and verified their correctness with QuickCheck.

To build the indexes, we use a *groupBy* function built from primitive functions *foldMapGroup* on bags and *singleton* for bags and maps respectively. While computing the indexes with *groupBy* is self-maintainable, merging them is not. We need to cache and incrementally update the intermediately created indexes to avoid recomputing them.

```
type Order = (ℤ, ℤ)
type LineItem = (ℤ, ℤ)

totalPrice :: Bag Order → Bag LineItem → ℤ
totalPrice orders lineItems = let
  orderIndex = groupBy fst orders
  orderSumIndex = Map.map (Bag.foldMapGroup snd) orderIndex
  lineItemIndex = groupBy fst lineItems
  lineItemSumIndex = Map.map (Bag.foldMapGroup snd) lineItemIndex
  merged = Map.merge orderSumIndex lineItemSumIndex
  total = Map.foldMapGroup multiply merged
  in total
```

This example is inspired by Koch et al. [2014]. Unlike them, we don't automate indexing, so to get good performance we start from a program that explicitly uses indexes.

We evaluate the performace in the same way we did for the average computation on bags. The initial input of size *n* is a pair of bags where both contain the pairs (i, i) for i between 1 and n. The sequence of changes are alternations between insertion and deletion of pairs (j, j) for different j. We alternate the insertions and deletions between the orders bag and the line items bag.

Our CTS derivative of the original program produces updated results much faster than from scratch recomputation and again the speedup increases with the size of the initial bags. For an initial size of the two bags of 800 incrementally updating the result is 180 times faster than from scratch recomputation.

17.5 Limitations and future work

In this section we describe limitations to be addressed in future work.

17.5.1 Hiding the cache type

In our experiments, functions of the same type $f_1, f_2 :: A \to B$ can be transformed to CTS functions $f_1 :: A \to (B, C_1), f_2 :: A \to (B, C_2)$ with different cache types C_1, C_2, since cache types depend on the implementation. We can fix this problem with some runtime overhead by using a single cache type *Cache*, defined as a tagged union of all cache types. If we defunctionalize function changes, we can index this cache type with tags representing functions, but other approaches are possible and we omit details. We conjecture (but have not proven) this fix gives a type-preserving translation, but leave this question for future work.

17.5.2 Nested bags

Our implementation of bags makes nested bags overly slow: we represent bags as tree-based maps from elements to multiplicity, so looking up a bag b in a bag of bags takes time proportional to the size of b. Possible solutions in this case include shredding, like done by [Koch et al., 2016]. We have no such problem for nested sequences, or other nested data which can be addressed in $O(1)$.

17.5.3 Proper tail calls

CTS transformation conflicts with proper tail calls, as it turns most tail calls into non-tail calls. In *A′NF* syntax, tail calls such as **let** $y = f\ x$ **in** $g\ y$ become **let** $y = f\ x$ **in let** $z = g\ y$ **in** z, and in CTS that becomes **let** $(y, c_y) = f\ x$ **in let** $(z, c_z) = g\ y$ **in** $(z, (c_y, c_z))$, where the call to g is genuinely *not* in tail position. This prevents recursion on deeply nested data structures like long lists: but such programs incrementalize inefficiently if deeply nested data is affected, so it is advisable to replace lists by other sequences anyway. It's unclear whether such fixes are available for other uses of tail calls.

17.5.4 Pervasive replacement values

Thanks to replacement changes, we can compute a change from any v_1 to any v_2 in constant time. Cai et al. [2014] use a difference operator $\ominus$ instead, but it's hard to implement $\ominus$ in constant time on values of non-constant size. So our formalization and implementation allow replacement values everywhere to ensure all computations can be incrementalized in some sense. Supporting replacement changes introduces overhead even if they are not used, because it prevents writing self-maintainable CTS derivatives. Hence, to improve performance, one should consider dropping support for replacement values and restricting supported computations. Consider a call to a binary CTS derivative $dfc\ da\ db\ c_1$ after computing $(y_1, c_1) = fc\ a_1\ b_1$: if db is a replacement change $!b_2$, then dfc must compute a result afresh by invoking $fc\ a_2\ b_2$, and $a_2 = a_1 \oplus da$ requires remembering previous input a_1 inside c_1. By the same argument, c_1 must also remember input b_1. Worse, replacement values are only needed to handle cases where incremental computation reduces to recomputation, because the new input is completely different, or because the condition of an **if** expression changed. Such changes to conditions are forbidden by other works [Koch et al., 2016], we believe for similar reasons.

17.5.5 Recomputing updated values

In some cases, the same updated input might be recomputed more than once. If a derivative df needs some base input x (that is, if df is not self-maintainable), df's input cache will contain a copy of x, and df's output cache will contain its updated value $x \oplus dx$. When all or most derivatives are self-maintainable this is convenient, because in most cases updated inputs will not need to be computed. But if most derivatives are not self-maintainable, the same updated input might be computed multiple times: specifically, if derivative dh calls functions df and dg, and both df and dg need the same base input x, caches for both df and dg will contain the updated value of $x \oplus dx$, computed independently. Worse, because of pervasive replacement values (Sec. 17.5.4), derivatives in our case studies tend to not be self-maintainable.

In some cases, such repeated updates should be removable by a standard optimizer after inlining and common-subexpression elimination, but it is unclear how often this happens. To solve this problem, derivatives could take and return both old inputs x_1 and updated ones $x_2 = x_1 \oplus dx$, and x_2 could be computed at the single location where dx is bound. In this case, to avoid updates for unused base inputs we would have to rely more on absence analysis (Sec. 17.5.6); pruning function

inputs appears easier than pruning caches. Otherwise, computations of updated inputs that are not used, in a lazy context, might cause space leaks, where thunks for $x_2 = x_1 \oplus dx_1$, $x_3 = x_2 \oplus dx_2$ and so on might accumulate and grow without bounds.

17.5.6 Cache pruning via absence analysis

To reduce memory usage and runtime overhead, it should be possible to automatically remove from transformed programs any caches or cache fragments that are not used (directly or indirectly) to compute outputs. Liu [2000] performs this transformation on CTS programs by using *absence analysis*, which was later extended to higher-order languages by Sergey et al. [2014]. In lazy languages, absence analysis removes thunks that are not needed to compute the output. We conjecture that, as long as the analysis is extended to *not* treat caches as part of the output, it should be able to remove unused caches or inputs (assuming unused inputs exist, see Sec. 17.5.4).

17.5.7 Unary vs n-ary abstraction

We only show our transformation correct for unary functions and tuples. But many languages provide efficient support for applying curried functions such as $div :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$, via either the *push-enter* or *eval-apply* evaluation model. For instance, invoking $div\ \ m\ \ n$ should not require the overhead of a function invocation for each argument [Marlow and Peyton Jones, 2006]. Naively transforming such a curried functions to CTS would produce a function *divc* of type $\mathbb{Z} \to (\mathbb{Z} \to (\mathbb{Z}, DivC_2)), DivC_1)$ with $DivC_1 = ()$, which adds excessive overhead.[3] Based on preliminary experiments, we believe we can straightforwardly combine our approach with a push-enter or eval-apply evaluation strategy for transformed curried functions. Alternatively, we expect that translating Haskell to Strict Core [Bolingbroke and Peyton Jones, 2009] would take care of turning Haskell into a language where function types describe their arity, which our transformation can easily take care of. We leave a more proper investigation for future work.

17.6 Related work

Of all research on incremental computation in both programming languages and databases [Gupta and Mumick, 1999; Ramalingam and Reps, 1993], we discuss the most closely related works. Other related work, more closely to cache-transfer style, is discussed in Chapter 19.

Previous work on cache-transfer-style Liu [2000]'s work has been the fundamental inspiration to this work, but her approach has no correctness proof and is restricted to a first-order untyped language (in part because no absence analysis for higher-order languages was available). Moreover, while the idea of cache-transfer-style is similar, it's unclear if her approach to incrementalization would extend to higher-order programs.

Firsov and Jeltsch [2016] also approach incrementalization by code transformation, but their approach does not deal with changes to functions. Instead of transforming functions written in terms of primitives, they provide combinators to write CTS functions and derivatives together. On the other hand, they extend their approach to support mutable caches, while restricting to immutable ones as we do might lead to a logarithmic slowdown.

[3] In Sec. 17.2 and our evaluation we use curried functions and never need to use this naive encoding, but only because we always invoke functions of known arity.

Finite differencing Incremental computation on collections or databases by finite differencing has a long tradition [Paige and Koenig, 1982; Blakeley et al., 1986]. The most recent and impressive line of work is the one on DBToaster [Koch, 2010; Koch et al., 2014], which is a highly efficient approach to incrementalize queries over bags by combining iterated finite differencing with other program transformations. They show asymptotic speedups both in theory and through experimental evaluations. Changes are only allowed for datatypes that form groups (such as bags or certain maps), but not for instance for lists or sets. Similar ideas were recently extended to higher-order and nested computation [Koch et al., 2016], though still restricted to datatypes that can be turned into groups.

Logical relations To study correctness of incremental programs we use a logical relation among initial values v_1, updated values v_2 and changes dv. To define a logical relation for an untyped λ-calculus we use a *step-indexed* logical relation, following [Appel and McAllester, 2001; Ahmed, 2006]; in particular, our definitions are closest to the ones by Acar et al. [2008], who also works with an untyped language, big-step semantics and (a different form of) incremental computation. However, their logical relation does not mention changes explicitly, since they do not have first-class status in their system. Moreover, we use environments rather than substitution, and use a slightly different step-indexing for our semantics.

Dynamic incrementalization The approaches to incremental computation with the widest applicability are in the family of self-adjusting computation [Acar, 2005, 2009], including its descendant Adapton [Hammer et al., 2014]. These approaches incrementalize programs by combining memoization and change propagation: after creating a trace of base computations, updated inputs are compared with old ones in $O(1)$ to find corresponding outputs, which are updated to account for input modifications. Compared to self-adjusting computation, Adapton only updates results when they are demanded. As usual, incrementalization is not efficient on arbitrary programs. To incrementalize efficiently a program must be designed so that input changes produce small changes to the computation trace; refinement type systems have been designed to assist in this task [Çiçek et al., 2016; Hammer et al., 2016]. Instead of comparing inputs by pointer equality, Nominal Adapton [Hammer et al., 2015] introduces first-class labels to identify matching inputs, enabling reuse in more situations.

Recording traces has often significant overheads in both space and time (slowdowns of 20-30× are common), though Acar et al. [2010] alleviate that by with datatype-specific support for tracing higher-level operations, while Chen et al. [2014] reduce that overhead by optimizing traces to not record redundant entries, and by logging chunks of operations at once, which reduces memory overhead but also potential speedups.

17.7 Chapter conclusion

We have presented a program transformation which turns a functional program into its derivative and efficiently shares redundant computations between them thanks to a statically computed cache. Although our first practical case studies show promising results, it remains now to integrate this transformation into a realistic compiler.

Chapter 18

Towards differentiation for System F

Differentiation is closely related to both logical relations and parametricity, as noticed by various authors in discussions of differentiation. Appendix C presents novel proofs of ILC correctness by adapting some logical relation techniques. As a demonstration, we define in this chapter differentiation for System F, by adapting results about parametricity. We stop short of a full proof that this generalization is correct, but we have implemented and tested it on a mature implementation of a System F typechecker; we believe the proof will mostly be a straightforward adaptation of existing ones about parametricity, but we leave verification for future work. A key open issue is discussed in Remark 18.2.1.

History and motivation Various authors noticed that differentiation appears related to (binary) parametricity (including Atkey [2015]). In particular, it resembles a transformation presented by Bernardy and Lasson [2011] for arbitrary PTSs.[1] By analogy with unary parametricity, Yufei Cai sketched an extension of differentiation for arbitrary PTSs, but many questions remained open, and at the time our correctness proof for ILC was significantly more involved and trickier to extend to System F, since it was defined in terms of denotational equivalence. Later, we reduced the proof core to defining a logical relation, proving its fundamental property and a few corollaries, as shown in Chapter 12, Extending this logical relation to System F proved comparably more straightforward.

Parametricity versus ILC Both parametricity and ILC define logical relations across program executions on different inputs. When studying parametricity, differences are only allowed in the implementations of abstractions (through abstract types or other mechanisms). To be related, different implementations of the same abstraction must give results that are equivalent according to the calling program. Indeed, parametricity defines not just a logical relation but a *logical equivalence*, that can be shown to be equivalent to contextual equivalence (as explained for instance by Harper [2016, Ch. 48] or by Ahmed [2006]).

When studying ILC, logical equivalence between terms t_1 and t_2 (written $(t_1, t_2) \in \mathcal{P}[\![\tau]\!]$), appears to be generalized by the existence of a valid change de between t_1 and t_2 (that is, $dt \triangleright t_1 \hookrightarrow$

[1] Bernardy and Lasson were not the first to introduce such a transformation. But most earlier work focuses on System F, and our presentation follows theirs and uses their added generality. We refer for details to existing discussions of related work [Wadler, 2007; Bernardy and Lasson, 2011].

$t_2 : \tau$). As in earlier chapters, if terms e_1 and e_2 are equivalent, any valid change dt between them is a nil change, but validity allows describing other changes.

18.1 The parametricity transformation

First, we show a variant of their parametricity transformation, adapted to a variant of STLC without base types but with type variables. Presenting $\lambda_{\to}$ using type variables will help when we come back to System F, and allows discussing parametricity on STLC. This transformation is based on the presentation of STLC as calculus $\lambda_{\to}$, a *Pure Type System* (PTS) [Barendregt, 1992, Sec. 5.2].

Background In PTSs, terms and types form a single syntactic category, but are distinguished through an extended typing judgement (written $\Gamma \vdash t : t$) using additional terms called *sorts*. Function types $\sigma \to \tau$ are generalized by Π-type $\Pi x : s.\, t$, where x can in general appear free in t, a generalization of Π-types in dependently-typed languages, but also of universal types $\forall X.\, T$ in the System F family; if x does not appear free in t, we write $s \to t$. Typing restricts whether terms of some sort s_1 can abstract over terms of sort s_2; different PTSs allow different combinations of sorts s_1 and s_2 (specified by *relations* $\mathcal{R}$), but lots of metatheory for PTSs is parameterized over the choice of combinations. In STLC presented as a PTS, there is an additional sort $\star$; those terms τ such that $\vdash \tau : \star$ are types.[2] We do not intend to give a full introduction to PTSs, only to introduce what's strictly needed for our presentation, and refer the readers to existing presentations [Barendregt, 1992].

Instead of base types, $\lambda_{\to}$ use uninterpreted type variables α, but do not allow terms to bind them. Nevertheless, we can write open terms, free in type variables for, say, naturals, and term variables for operations on naturals. STLC restricts Π-types $\Pi x : A.\, B$ to the usual arrow types $A \to B$ through $\mathcal{R}$: one can show that in $\Pi x : A.\, B$, variable x cannot occur free in B.

Parametricity Bernardy and Lasson show how to transform a typed term $\Gamma \vdash t : \tau$ in a strongly normalizing PTS P into a proof that t satisfies a parametricity statement for τ. The proof is in a logic represented by PTS P^2. PTS P^2 is constructed uniformly from P, and is strongly normalizing whenever P is. When the base PTS P is $\lambda_{\to}$, Bernardy and Lasson's transformation produces terms in a PTS $\lambda^2_{\to}$, produced by transforming $\lambda_{\to}$. PTS $\lambda^2_{\to}$ extends $\lambda_{\to}$ with a separate sort $\lceil\star\rceil$ of propositions, together with enough abstraction power to abstract propositions over values. Like $\lambda_{\to}$, $\lambda^2_{\to}$ uses uninterpreted type variables α and does not allow abstracting over them.

In parametricity statements about $\lambda_{\to}$, we write $(t_1, t_2) \in \mathcal{P}\llbracket\, \tau \,\rrbracket$ for a proposition (hence, living in $\lceil\star\rceil$) that states that t_1 and t_2 are related. This proposition is defined, as usual, via a *logical relation*. We write dxx for a proof that x_1 and x_2 are related. For type variables α, transformed terms abstract over an arbitrary relation $\mathcal{R}^\alpha$ between α_1 and α_2. When α is instantiated by τ, $\mathcal{R}^\alpha$ *can* (but does not have to) be instantiated with relation $(-, -) \in \mathcal{P}\llbracket\, \tau \,\rrbracket$, but $\mathcal{R}^\alpha$ abstracts over arbitrary *candidate* relations (similar to the notion of reducibility candidates [Girard et al., 1989, Ch. 14.1.1]). Allowing alternative instantiations makes parametricity statements more powerful, and it is also necessary to define parametricity for impredicative type systems (like System F) in a predicative ambient logic.[3]

A transformed term $\mathcal{P}\llbracket\, t \,\rrbracket$ relates two executions of terms t in different environments: it can be read as a proof that term t maps related inputs to related outputs. The proof strategy that $\mathcal{P}\llbracket\, t \,\rrbracket$ uses is the standard one for proving fundamental properties of logical relations; but instead of

[2] Calculus $\lambda_{\to}$ has also a sort $\Box$ such that $\vdash \star : \Box$, but $\Box$ itself has no type. Such a sort appears only in PTS typing derivations, not in well-typed terms themselves, so we do not discuss it further.

[3] Specifically, if we require $\mathcal{R}^\alpha$ to be instantiated with the logical relation itself for τ when α is instantiated with τ, the definition becomes circular. Consider the definition of $(t_1, t_2) \in \mathcal{P}\llbracket\, \forall\alpha.\, \tau \,\rrbracket$: type variable α can be instantiated with the same type $\forall\alpha.\, \tau$, so the definition of $(t_1, t_2) \in \mathcal{P}\llbracket\, \forall\alpha.\, \tau \,\rrbracket$ becomes impredicative.

doing a proof in the metalevel logic (by induction on terms and typing derivations), $\lambda^2_\to$ serves as an object-level logic, and $\mathcal{P}\llbracket - \rrbracket$ generates proof terms in this logic by recursion on terms. At the metalevel, Bernardy and Lasson, Th. 3 prove that for any well-typed term $\Gamma \vdash t : \tau$, proof $\mathcal{P}\llbracket t \rrbracket$ shows that t satisfies the parametricity statement for type τ given the hypotheses $\mathcal{P}\llbracket \Gamma \rrbracket$ (or more formally, $\mathcal{P}\llbracket \Gamma \rrbracket \vdash \mathcal{P}\llbracket t \rrbracket : (\mathcal{S}_1\llbracket t \rrbracket, \mathcal{S}_2\llbracket t \rrbracket) \in \mathcal{P}\llbracket \tau \rrbracket$).

$$
\begin{aligned}
&(f_1, f_2) \in \mathcal{P}\llbracket \sigma \to \tau \rrbracket = \Pi(x_1 : \mathcal{S}_1\llbracket \sigma \rrbracket)\ (x_2 : \mathcal{S}_2\llbracket \sigma \rrbracket)\ (dxx : (x_1, x_2) \in \mathcal{P}\llbracket \sigma \rrbracket).\\
&\quad (f_1\ x_1, f_2\ x_2) \in \mathcal{P}\llbracket \tau \rrbracket\\
&(x_1, x_2) \in \mathcal{P}\llbracket \alpha \rrbracket = \mathcal{R}^\alpha\ x_1\ x_2\\
&\mathcal{P}\llbracket x \rrbracket = dxx\\
&\mathcal{P}\llbracket \lambda(x : \sigma) \to t \rrbracket =\\
&\quad \lambda(x_1 : \mathcal{S}_1\llbracket \sigma \rrbracket)\ (x_2 : \mathcal{S}_2\llbracket \sigma \rrbracket)\ (dxx : (x_1, x_2) \in \mathcal{P}\llbracket \sigma \rrbracket) \to\\
&\qquad \mathcal{P}\llbracket t \rrbracket\\
&\mathcal{P}\llbracket s\ t \rrbracket = \mathcal{P}\llbracket s \rrbracket\ \mathcal{S}_1\llbracket t \rrbracket\ \mathcal{S}_2\llbracket t \rrbracket\ \mathcal{P}\llbracket t \rrbracket\\
&\mathcal{P}\llbracket \varepsilon \rrbracket = \varepsilon\\
&\mathcal{P}\llbracket \Gamma, x : \tau \rrbracket = \mathcal{P}\llbracket \Gamma \rrbracket, x_1 : \mathcal{S}_1\llbracket \tau \rrbracket, x_2 : \mathcal{S}_2\llbracket \tau \rrbracket, dxx : (x_1, x_2) \in \mathcal{P}\llbracket \tau \rrbracket\\
&\mathcal{P}\llbracket \Gamma, \alpha : \star \rrbracket = \mathcal{P}\llbracket \Gamma \rrbracket, \alpha_1 : \star, \alpha_2 : \star, \mathcal{R}^\alpha : \alpha_1 \to \alpha_2 \to \lceil\star\rceil
\end{aligned}
$$

In the above, $\mathcal{S}_1\llbracket s \rrbracket$ and $\mathcal{S}_2\llbracket s \rrbracket$ simply subscript all (term and type) variables in their arguments with $_1$ and $_2$, to make them refer to the first or second computation. To wit, the straightforward definition is:

$$
\begin{aligned}
&\mathcal{S}_i\llbracket x \rrbracket = x_i\\
&\mathcal{S}_i\llbracket \lambda(x : \sigma) \to t \rrbracket = \lambda(x_i : \mathcal{S}_i\llbracket \sigma \rrbracket) \to \mathcal{S}_i\llbracket t \rrbracket\\
&\mathcal{S}_i\llbracket s\ t \rrbracket = \mathcal{S}_i\llbracket s \rrbracket\ \mathcal{S}_i\llbracket t \rrbracket\\
&\mathcal{S}_i\llbracket \sigma \to \tau \rrbracket = \mathcal{S}_i\llbracket \sigma \rrbracket \to \mathcal{S}_i\llbracket \tau \rrbracket\\
&\mathcal{S}_i\llbracket \alpha \rrbracket = \alpha_i
\end{aligned}
$$

It might be unclear how the proof $\mathcal{P}\llbracket t \rrbracket$ references the original term t. Indeed, it does not do so explicitly. But since in PTS β-equivalent types have the same members, we can construct typing judgement that mention t. As mentioned, from $\Gamma \vdash t : \tau$ it follows that $\mathcal{P}\llbracket \Gamma \rrbracket \vdash \mathcal{P}\llbracket t \rrbracket : (\mathcal{S}_1\llbracket t \rrbracket, \mathcal{S}_2\llbracket t \rrbracket) \in \mathcal{P}\llbracket \tau \rrbracket$ [Bernardy and Lasson, 2011, Th. 3]. This is best shown on a small example.

Example 18.1.1
Take for instance an identity function $id = \lambda(x : \alpha) \to x$, which is typed in an open context (that is, $\alpha : \star \vdash \lambda(x : \alpha) \to x$). The transformation gives us

$$pid = \mathcal{P}\llbracket id \rrbracket = \lambda(x_1 : \alpha_1)\ (x_2 : \alpha_2)\ (dxx : (x_1, x_2) \in \mathcal{P}\llbracket \alpha \rrbracket) \to dxx,$$

which simply returns the proofs that inputs are related:

$$\alpha_1 : \star, \alpha_2 : \star, \mathcal{R}^\alpha : \alpha_1 \to \alpha_2 \to \lceil\star\rceil \vdash pid : \Pi(x_1 : \alpha_1)\ (x_2 : \alpha_2).\ (x_1, x_2) \in \mathcal{P}\llbracket \alpha \rrbracket \to (x_1, x_2) \in \mathcal{P}\llbracket \alpha \rrbracket\,.$$

This typing judgement does not mention id. But since $x_1 =_\beta \mathcal{S}_1\llbracket id \rrbracket\ x_1$ and $x_2 = \mathcal{S}_2\llbracket id \rrbracket\ x_2$, we can also show that id's outputs are related whenever the inputs are:

$$\alpha_1 : \star, \alpha_2 : \star, \mathcal{R}^\alpha : \alpha_1 \to \alpha_2 \to \lceil\star\rceil \vdash pid : \Pi(x_1 : \alpha_1)\ (x_2 : \alpha_2).\ (x_1, x_2) \in \mathcal{P}\llbracket \alpha \rrbracket \to (\mathcal{S}_1\llbracket id \rrbracket\ x_1, \mathcal{S}_2\llbracket id \rrbracket\ x_2) \in \mathcal{P}\llbracket \alpha \rrbracket\,.$$

More concisely, we can show that:

$$\alpha_1 : \star, \alpha_2 : \star, \mathcal{R}^\alpha : \alpha_1 \to \alpha_2 \to \lceil\star\rceil \vdash pid : (\mathcal{S}_1\llbracket id \rrbracket, \mathcal{S}_2\llbracket id \rrbracket) \in \mathcal{P}\llbracket \alpha \to \alpha \rrbracket\,.$$

□

18.2 Differentiation and parametricity

By altering the transformation for binary parametricity, we obtain a variant of differentiation close to a parametricity transformation investigated by Bernardy, Jansson, and Paterson [2010]. Instead of only having proofs that values are related, we can modify $(t_1, t_2) \in \mathcal{P}\,[\![\,\tau\,]\!]$ to be a type of values — more precisely, a dependent type $(t_1, t_2) \in \Delta_2\,[\![\,\tau\,]\!]$ of valid changes, indexed by source $t_1 : \mathcal{S}_1\,[\![\,\tau\,]\!]$ and destination $t_2 : \mathcal{S}_2\,[\![\,\tau\,]\!]$. Similarly, $\mathcal{R}^\alpha$ is replaced by a dependent type of changes, not propositions, that we write Δ_α. Formally, this is encoded by replacing sort $\lceil\star\rceil$ with $\star$ in Bernardy and Lasson [2011]'s transform, and by moving target programs in a PTS that allows for both simple and dependent types, like the Edinburgh Logical Framework; such a PTS is known as λ_P [Barendregt, 1992].

For type variables α, we specialize the transformation further, ensuring that $\alpha_1 = \alpha_2 = \alpha$ and adapting $\mathcal{S}_1\,[\![\,-\,]\!], \mathcal{S}_2\,[\![\,-\,]\!]$ accordingly to *not* add subscripts to type variable:

$$\mathcal{S}_i\,[\![\,\alpha\,]\!] = \alpha$$

Without this specialization, we get to deal with changes across different types, which we don't do just yet but defer to Sec. 18.4.

We first show how the transformation affects typing contexts:

$$\begin{aligned}
&\mathcal{D}\,[\![\,\varepsilon\,]\!] = \varepsilon \\
&\mathcal{D}\,[\![\,\Gamma, x : \sigma\,]\!] = \mathcal{D}\,[\![\,\Gamma\,]\!], x_1 : \sigma, x_2 : \sigma, dx : (x_1, x_2) \in \Delta_2\,[\![\,\sigma\,]\!] \\
&\mathcal{D}\,[\![\,\Gamma, \alpha : \star\,]\!] = \mathcal{D}\,[\![\,\Gamma\,]\!], \alpha : \star, \Delta_\alpha : \alpha \to \alpha \to \star
\end{aligned}$$

Unlike standard differentiation, transformed programs bind, for each input variable $x : \sigma$, a source $x_1 : \sigma$, a destination $x_2 : \sigma$, and a valid change dx from x_1 to x_2. More in detail, for each variable in input programs $x : \sigma$, programs produced by standard differentiation bind both $x : \sigma$ and $dx : \sigma$; valid use of these programs requires dx to be a valid change with source x, but this is not enforced through types. Instead, programs produced by *this variant* of differentiation bind, for each variable in input programs $x : \sigma$, both source $x_1 : \sigma$, destination $x_2 : \sigma$, and a valid change dx from x_1 to x_2, where validity is enforced through dependent types.

For input type variables α, output programs also bind a dependent type of valid changes Δ_α.

Next, we define the type of changes. Since change types are indexed by source and destination, the type of function changes forces them to be valid, and the definition of function changes resembles our earlier logical relation for validity. More precisely, if df is a change from f_1 to f_2 ($df : (f_1, f_2) \in \Delta_2\,[\![\,\sigma \to \tau\,]\!]$), then df must map any change dx from initial input x_1 to updated input x_2 to a change from initial output $f_1\ x_1$ to updated output $f_2\ x_2$.

$$\begin{aligned}
&(f_1, f_2) \in \Delta_2\,[\![\,\sigma \to \tau\,]\!] = \Pi(x_1\ x_2 : \sigma)\ (dx : (x_1, x_2) \in \Delta_2\,[\![\,\sigma\,]\!]). \\
&\quad (f_1\ x_1, f_2\ x_2) \in \Delta_2\,[\![\,\tau\,]\!] \\
&(x_1, x_2) \in \Delta_2\,[\![\,\alpha\,]\!] = \Delta_\alpha\ x_1\ x_2
\end{aligned}$$

At this point, the transformation on terms acts on abstraction and application to match the transformation on typing contexts. The definition of $\mathcal{D}\,[\![\,\lambda(x : \sigma) \to t\,]\!]$ follows the definition of $\mathcal{D}\,[\![\,\Gamma, x : \sigma\,]\!]$ — both transformation results bind the same variables x_1, x_2, dx with the same types. Application provides corresponding arguments.

$$\begin{aligned}
&\mathcal{D}\,[\![\,x\,]\!] = dx \\
&\mathcal{D}\,[\![\,\lambda(x : \sigma) \to t\,]\!] = \lambda(x_1\ x_2 : \sigma)\ (dx : (x_1, x_2) \in \Delta_2\,[\![\,\sigma\,]\!]) \to \mathcal{D}\,[\![\,t\,]\!] \\
&\mathcal{D}\,[\![\,s\ t\,]\!] = \mathcal{D}\,[\![\,s\,]\!]\ \ \mathcal{S}_1\,[\![\,t\,]\!]\ \ \mathcal{S}_2\,[\![\,t\,]\!]\ \ \mathcal{D}\,[\![\,t\,]\!]
\end{aligned}$$

If we extend the language to support primitives and their manually-written derivatives, it is useful to have contexts also bind, next to type variables α, also change structures for α, to allow terms to use change operations. Since the differentiation output does not use change operations here, we omit change structures for now.

Remark 18.2.1 (Validity and $\oplus$)
Because we omit change structures (here and through most of the chapter), the type of differentiation only suggests that $\mathcal{D}[\![t]\!]$ is a valid change, but not that validity agrees with $\oplus$.

In other words, dependent types as used here do not prove all theorems expected of an incremental transformation. While we can prove the fundamental property of our logical relation, we cannot prove that $\oplus$ agrees with differentiation for abstract types. As we did in Sec. 13.4 (in particular Plugin Requirement 13.4.1), we must require that validity relations provided for type variables agree with implementations of $\oplus$ as provided for type variables: formally, we must state (internally) that $\forall(x_1\ x_2 : \alpha)\ (dx : \Delta_\alpha\ x_1\ x_2).\ (x_1 \oplus dx, x_2) \in \mathcal{P}[\![\alpha]\!]$. We can state this requirement by internalizing logical equivalence (such as shown in Sec. 18.1, or using Bernardy et al. [2010]'s variant in λ_P). But since this statement quantifies over members of α, it is not clear if it can be proven internally when instantiating α with a type argument. We leave this question (admittedly crucial) for future work.□

This transformation is not incremental, as it recomputes both source and destination for each application, but we could fix this by replacing $\mathcal{S}_2[\![s]\!]$ with $\mathcal{S}_1[\![s]\!] \oplus \mathcal{D}[\![s]\!]$ (and passing change structures to make $\oplus$ available to terms), or by not passing destinations. We ignore such complications here.

18.3 Proving differentiation correct externally

Instead of producing dependently-typed programs that show their correctness, we might want to produce simply-typed programs (which are easier to compile efficiently) and use an external correctness proof, as in Chapter 12. We show how to generate such external proofs here. For simplicity, here we produce external correctness proofs for the transformation we just defined on dependently-typed outputs, rather than defining a separate simply-typed transformation.

Specifically, we show how to generate from well-typed terms t a proof that $\mathcal{D}[\![t]\!]$ is correct, that is, that $(\mathcal{S}_1[\![t]\!], \mathcal{S}_2[\![t]\!], \mathcal{D}[\![t]\!]) \in \Delta\mathcal{V}[\![\tau]\!]$.

Proofs are generated through the following transformation, defined in terms of the transformation from Sec. 18.2. Again, we show first typing contexts and the logical relation:

$$
\begin{aligned}
&\mathcal{DP}[\![\varepsilon]\!] = \varepsilon \\
&\mathcal{DP}[\![\Gamma, x : \tau]\!] = \mathcal{DP}[\![\Gamma]\!], x_1 : \tau, x_2 : \tau, \\
&\quad dx : (x_1, x_2) \in \Delta_2[\![\tau]\!], dxx : (x_1, x_2, dx) \in \Delta\mathcal{V}[\![\tau]\!] \\
&\mathcal{DP}[\![\Gamma, \alpha : \star]\!] = \mathcal{DP}[\![\Gamma]\!], \alpha : \star, \\
&\quad \Delta_\alpha : \alpha \to \alpha \to \star, \\
&\quad \mathcal{R}^\alpha : \Pi(x_1 : \alpha)\ (x_2 : \alpha)\ (dx : (x_1, x_2) \in \Delta_2[\![\alpha]\!]) \to \lceil\star\rceil \\
&(f_1, f_2, df) \in \Delta\mathcal{V}[\![\sigma \to \tau]\!] = \\
&\quad \Pi(x_1\ x_2 : \sigma)\ (dx : (x_1, x_2) \in \Delta_2[\![\sigma]\!])\ (dxx : (x_1, x_2, dx) \in \Delta\mathcal{V}[\![\sigma]\!]). \\
&\qquad (f_1\ x_1, f_2\ x_2, df\ x_1\ x_2\ dx) \in \Delta\mathcal{V}[\![\tau]\!] \\
&(x_1, x_2, dx) \in \Delta\mathcal{V}[\![\alpha]\!] = \mathcal{R}^\alpha\ x_1\ x_2\ dx
\end{aligned}
$$

The transformation from terms to proofs then matches the definition of typing contexts:

$$
\begin{aligned}
&\mathcal{DP}[\![x]\!] = dxx \\
&\mathcal{DP}[\![\lambda(x : \sigma) \to t]\!] =
\end{aligned}
$$

$$\lambda(x_1\ x_2 : \sigma)\ (dx : (x_1, x_2) \in \Delta_2\ [\![\ \sigma\ ]\!])\ (dxx : (x_1, x_2, dx) \in \Delta\mathcal{V}\ [\![\ \sigma\ ]\!]) \to \mathcal{DP}\ [\![\ t\ ]\!]$$
$$\mathcal{DP}\ [\![\ s\ t\ ]\!] = \mathcal{DP}\ [\![\ s\ ]\!]\ \mathcal{S}_1\ [\![\ t\ ]\!]\ \mathcal{S}_2\ [\![\ t\ ]\!]\ \mathcal{D}\ [\![\ t\ ]\!]\ \mathcal{DP}\ [\![\ t\ ]\!]$$

This term produces a proof object in PTS λ_P^2, which is produced by augmenting λ_P following Bernardy and Lasson [2011]. The informal proof content of generated proofs follows the proof of $\mathcal{D}\ [\![-]\!]$'s correctness (Theorem 12.2.2): For a variable x, we just use the assumption dxx that $(x_1, dx, x_2) \in \Delta\mathcal{V}\ [\![\ \tau\ ]\!]$, that we have in context. For abstractions $\lambda x \to t$, we have to show that $\mathcal{D}\ [\![\ t\ ]\!]$ is correct for all valid input changes x_1, x_2, dx and for all proofs $dxx : (x_1, dx, x_2) \in \Delta\mathcal{V}\ [\![\ \sigma\ ]\!]$ that dx is indeed a valid input change, so we bind all those variables in context, including proof dxx, and use $\mathcal{DP}\ [\![\ t\ ]\!]$ recursively to prove that $\mathcal{D}\ [\![\ t\ ]\!]$ is correct in the extended context. For applications $s\ t$, we use the proof that $\mathcal{D}\ [\![\ s\ ]\!]$ is correct (obtained recursively via $\mathcal{DP}\ [\![\ s\ ]\!]$). To show that $\mathcal{D}\ [\![\ s\ ]\!]\ \mathcal{D}\ [\![\ t\ ]\!]$, that is $\mathcal{D}\ [\![\ s\ ]\!]\ \mathcal{S}_1\ [\![\ s\ ]\!]\ \mathcal{S}_2\ [\![\ s\ ]\!]\ \mathcal{D}\ [\![\ t\ ]\!]$, we simply show that $\mathcal{D}\ [\![\ s\ ]\!]$ is being applied to valid inputs, using the proof that $\mathcal{D}\ [\![\ t\ ]\!]$ is correct (obtained recursively via $\mathcal{DP}\ [\![\ t\ ]\!]$).

18.4 Combinators for polymorphic change structures

Earlier, we restricted our transformation on $\lambda_\to$ terms, so that there could be a change dt from t_1 to t_2 only if t_1 and if t_2 have the same type. In this section we lift this restriction and define *polymorphic* change structures (also called change structures when no ambiguity arises). To do so, we sketch how to extend core change-structure operations to polymorphic change structures.

We also show a few *combinators*, combining existing polymorphic change structures into new ones. We believe the combinator types are more enlightening than their implementations, but we include them here for completeness. We already described a few constructions on non-polymorphic change structures; however, polymorphic change structures enable new constructions that insert or remove data, or apply isomorphisms to the source or destination type.

We conjecture that combinators for polymorphic change structure can be used to compose, out of small building blocks, change structures that, for instance, allow inserting or removing elements from a recursive datatype such as lists. Derivatives for primitives could then be produced using equational reasoning as described in Sec. 11.2. However, we leave investigation of such avenues to future work.

We describe change operations for polymorphic change structures via a Haskell record containing change operations, and we define combinators on polymorphic change structures. Of all change operations, we only consider $\oplus$; to avoid confusion, we write $\boxplus$ for the polymorphic variant of $\oplus$.

```
data ℂ𝕊 τ₁ δ_τ τ₂ = ℂ𝕊 {
  (⊞) :: τ₁ → δ_τ → τ₂
}
```

This code define a record type constructor $\mathbb{CS}$ with a data constructor also written $\mathbb{CS}$, and a field accessor written $(\boxplus)$; we use τ_1, δ_τ and τ_2 (and later also σ_1, δ_σ and σ_2) for Haskell type variables. To follow Haskell lexical rules, we use here lowercase letters (even though Greek ones).

We have not formalized definitions of validity, or proofs that it agrees with $\boxplus$, but for all the change structures and combinators in this section, this exercise appears no harder than the ones in Chapter 13.

In Sec. 11.1 and 17.2 change structures are embedded in Haskell using type class *ChangeStruct* τ. Conversely, here we do not define a type class of polymorphic change structures, because (apart from the simplest scenarios), Haskell type class resolution is unable to choose a *canonical* way to construct a polymorphic change structure using our combinators.

All existing change structures (that is, instances of *ChangeStruct* τ) induce generalized change structures $\mathbb{CS}\ \tau\ (\Delta\tau)\ \tau$.

```
typeCS :: ChangeStruct τ ⇒ ℂ𝕊 τ (Δτ) τ
typeCS = ℂ𝕊 (⊕)
```

We can also have change structures across different types. Replacement changes are possible:

```
replCS :: ℂ𝕊 τ1 τ2 τ2
replCS = ℂ𝕊 $ λx1 x2 → x2
```

But replacement changes are not the only option. For product types, or for any form of nested data, we can apply changes to the different components, changing the type of some components. We can also define a change structure for nullary products (the unit type) which can be useful as a building block in other change structures:

```
prodCS :: ℂ𝕊 σ1 δσ σ2 → ℂ𝕊 τ1 δτ τ2 → ℂ𝕊 (σ1, τ1) (δσ, δτ) (σ2, τ2)
prodCS scs tcs = ℂ𝕊 $ λ(s1, t1) (ds, dt) → ((⊞) scs s1 ds, (⊞) tcs t1 dt)
unitCS :: ℂ𝕊 () () ()
unitCS = ℂ𝕊 $ λ() () → ()
```

The ability to modify a field to one of a different type is also known as in the Haskell community as *polymorphic record update*, a feature that has proven desirable in the context of lens libraries [O'Connor, 2012; Kmett, 2012].

We can also define a combinator *sumCS* for change structures on sum types, similarly to our earlier construction described in Sec. 11.6. This time, we choose to forbid changes across branches since they're inefficient, though we could support them as well, if desired.

```
sumCS :: ℂ𝕊 s1 ds s2 → ℂ𝕊 t1 dt t2 →
  ℂ𝕊 (Either s1 t1) (Either ds dt) (Either s2 t2)
sumCS scs tcs = ℂ𝕊 go
  where
    go (Left s1) (Left ds) = Left $ (⊞) scs s1 ds
    go (Right t1) (Right dt) = Right $ (⊞) tcs t1 dt
    go _ _ = error "Invalid changes"
```

Given two change structures from τ_1 to τ_2, with respective change types $\delta_{\tau a}$ and $\delta_{\tau b}$, we can also define a new change structure with change type *Either* $\delta_{\tau a}\ \delta_{\tau b}$, that allows using changes from either structure. We capture this construction through combinator *mSumCS*, having the following signature:

```
mSumCS :: ℂ𝕊 τ1 δτa τ2 → ℂ𝕊 τ1 δτb τ2 → ℂ𝕊 τ1 (Either δτa δτb) τ2
mSumCS acs bcs = ℂ𝕊 go
  where
    go t1 (Left dt) = (⊞) acs t1 dt
    go t1 (Right dt) = (⊞) bcs t1 dt
```

This construction is possible for non-polymorphic change structures; we only need change structures to be first-class (instead of a type class) to be able to internalize this construction in Haskell.

Using combinator *lInsCS* we can describe updates going from type τ_1 to type (σ, τ_2), assuming a change structure from τ_1 to τ_2: that is, we can prepend a value of type σ to our data while we modify it. Similarly, combinator *lRemCS* allows removing values:

$$
\begin{aligned}
&\mathit{lInsCS} :: \mathbb{CS}\ t_1\ dt\ t_2 \to \mathbb{CS}\ t_1\ (s, dt)\ (s, t_2)\\
&\mathit{lInsCS}\ tcs = \mathbb{CS}\ \$\ \lambda t_1\ (s, dt) \to (s, (\boxplus)\ tcs\ t_1\ dt)\\
&\mathit{lRemCS} :: \mathbb{CS}\ t_1\ dt\ (s, t_2) \to \mathbb{CS}\ t_1\ dt\ t_2\\
&\mathit{lRemCS}\ tcs = \mathbb{CS}\ \$\ \lambda t_1\ dt \to \mathit{snd}\ \$\ (\boxplus)\ tcs\ t_1\ dt
\end{aligned}
$$

We can also transform change structures given suitable conversion functions.

$$
\begin{aligned}
&\mathit{lIsoCS} :: (t_1 \to s_1) \to \mathbb{CS}\ s_1\ dt\ t_2 \to \mathbb{CS}\ t_1\ dt\ t_2\\
&\mathit{mIsoCS} :: (dt \to ds) \to \mathbb{CS}\ t_1\ ds\ t_2 \to \mathbb{CS}\ t_1\ dt\ t_2\\
&\mathit{rIsoCS} :: (s_2 \to t_2) \to \mathbb{CS}\ t_1\ dt\ s_2 \to \mathbb{CS}\ t_1\ dt\ t_2\\
&\mathit{isoCS} :: (t_1 \to s_1) \to (dt \to ds) \to (s_2 \to t_2) \to \mathbb{CS}\ s_1\ ds\ s_2 \to \mathbb{CS}\ t_1\ dt\ t_2
\end{aligned}
$$

To do so, we must only compose $\boxplus$ with the given conversion functions according to the types. Combinator *isoCS* arises by simply combining the other ones:

$$
\begin{aligned}
&\mathit{lIsoCS}\ f\ tcs &&= \mathbb{CS}\ \$\ \lambda s_1\ dt \to (\boxplus)\ tcs\ (f\ s_1)\ dt\\
&\mathit{mIsoCS}\ g\ tcs &&= \mathbb{CS}\ \$\ \lambda t_1\ ds \to (\boxplus)\ tcs\ t_1\ (g\ ds)\\
&\mathit{rIsoCS}\ h\ tcs &&= \mathbb{CS}\ \$\ \lambda t_1\ dt \to h\ \$\ (\boxplus)\ tcs\ t_1\ dt\\
&\mathit{isoCS}\ f\ g\ h\ scs &&= \mathit{lIsoCS}\ f\ \$\ \mathit{mIsoCS}\ g\ \$\ \mathit{rIsoCS}\ h\ scs
\end{aligned}
$$

With a bit of datatype-generic programming infrastructure, we can reobtain only using combinators the change structure for lists shown in Sec. 11.3.3, which allows modifying list elements. The core definition is the following one:

$$
\begin{aligned}
&\mathit{listMuChangeCS} :: \mathbb{CS}\ a_1\ da\ a_2 \to \mathbb{CS}\ (\mathit{List}_\mu\ a_1)\ (\mathit{List}_\mu\ da)\ (\mathit{List}_\mu\ a_2)\\
&\mathit{listMuChangeCS}\ acs = go\ \textbf{where}\\
&\quad go = \mathit{isoCS}\ \mathit{unRollL}\ \mathit{unRollL}\ \mathit{rollL}\ \$\\
&\qquad\quad \mathit{sumCS}\ \mathit{unitCS}\ \$\ \mathit{prodCS}\ acs\ go
\end{aligned}
$$

The needed infrastructure appears in Fig. 18.1.

Section summary We have defined polymorphic change structures and shown they admit a rich combinator language. Now, we return to using these change structures for differentiating $\lambda_{\to}$ and System F.

18.5 Differentiation for System F

After introducing changes across different types, we can also generalize differentiation for $\lambda_{\to}$ so that it allows for the now generalized changes:

$$
\begin{aligned}
&(f_1, f_2) \in \Delta_2 [\![\sigma \to \tau]\!] = \Pi(x_1 : \mathcal{S}_1 [\![\sigma]\!])\ (x_2 : \mathcal{S}_2 [\![\sigma]\!])\ (dx : (x_1, x_2) \in \Delta_2 [\![\sigma]\!]).\\
&\quad (f_1\ x_1, f_2\ x_2) \in \Delta_2 [\![\tau]\!]\\
&(x_1, x_2) \in \Delta_2 [\![\alpha]\!] = \Delta_\alpha\ x_1\ x_2\\
&\mathcal{D} [\![x]\!] = dx\\
&\mathcal{D} [\![\lambda(x : \sigma) \to t]\!] = \lambda(x_1 : \mathcal{S}_1 [\![\sigma]\!])\ (x_2 : \mathcal{S}_2 [\![\sigma]\!])\ (dx : (x_1, x_2) \in \Delta_2 [\![\sigma]\!]) \to \mathcal{D} [\![t]\!]\\
&\mathcal{D} [\![s\ t]\!] = \mathcal{D} [\![s]\!]\ \mathcal{S}_1 [\![t]\!]\ \mathcal{S}_2 [\![t]\!]\ \mathcal{D} [\![t]\!]\\
&\mathcal{D} [\![\varepsilon]\!] = \varepsilon\\
&\mathcal{D} [\![\Gamma, x : \tau]\!] = \mathcal{D} [\![\Gamma]\!], x_1 : \mathcal{S}_1 [\![\tau]\!], x_2 : \mathcal{S}_2 [\![\tau]\!], dx : (x_1, x_2) \in \Delta_2 [\![\tau]\!]\\
&\mathcal{D} [\![\Gamma, \alpha : \star]\!] = \mathcal{D} [\![\Gamma]\!], \alpha_1 : \star, \alpha_2 : \star, \Delta_\alpha : \alpha_1 \to \alpha_2 \to \star
\end{aligned}
$$

```
  -- Our list type:
data List a = Nil | Cons a (List a)
  -- Equivalently, we can represent lists as a fixpoint of a sum-of-product pattern functor:
data Mu f = Roll { unRoll :: f (Mu f) }
data ListF a x = L { unL :: Either () (a, x) }
type List_μ a = Mu (ListF a)
rollL :: Either () (a, List_μ a) → List_μ a
rollL = Roll ∘ L
unRollL :: List_μ a → Either () (a, List_μ a)
unRollL = unL ∘ unRoll
  -- Isomorphism between List and List_μ:
lToLMu :: List a → List_μ a
lToLMu Nil = rollL $ Left ()
lToLMu (Cons a as) = rollL $ Right (a, lToLMu as)
lMuToL :: List_μ a → List a
lMuToL = go ∘ unRollL where
  go (Left ()) = Nil
  go (Right (a, as)) = Cons a (lMuToL as)
  -- A change structure for List:
listChangesCS :: ℂ𝕊 a1 da a2 → ℂ𝕊 (List a1) (List da) (List a2)
listChangesCS acs = isoCS lToLMu lToLMu lMuToL $ listMuChangeCS acs
```

Figure 18.1: A change structure that allows modifying list elements.

$$\mathcal{S}_i \llbracket \alpha \rrbracket = \alpha_i$$

By adding a few additional rules, we can extend differentiation to System F (the PTS λ_2). We choose to present the additional rules using System F syntax rather than PTS syntax.

$$\begin{aligned}
&(f_1, f_2) \in \Delta_2 \llbracket \forall \alpha.\, \tau \rrbracket = \\
&\quad \Pi(\alpha_1 : \star)\ (\alpha_2 : \star)\ (\Delta_\alpha : \alpha_1 \to \alpha_2 \to \star).\ (f_1\ [\alpha_1], f_2\ [\alpha_2]) \in \Delta_2 \llbracket \tau \rrbracket \\
&\mathcal{D} \llbracket \Lambda \alpha.\, t \rrbracket = \\
&\quad \lambda(\alpha_1\ \alpha_2 : \star)\ (\Delta_\alpha : \alpha_1 \to \alpha_2 \to \star) \to \mathcal{D} \llbracket t \rrbracket \\
&\mathcal{D} \llbracket t\ [\tau] \rrbracket = \mathcal{D} \llbracket t \rrbracket\ \mathcal{S}_1 \llbracket \tau \rrbracket\ \mathcal{S}_2 \llbracket \tau \rrbracket\ ((-,-) \in \Delta_2 \llbracket \tau \rrbracket) \\
&\mathcal{S}_i \llbracket \forall \alpha.\, \tau \rrbracket = \forall \alpha_i.\, \mathcal{S}_i \llbracket \tau \rrbracket \\
&\mathcal{S}_i \llbracket \Lambda \alpha.\, t \rrbracket = \Lambda \alpha_i.\, \mathcal{S}_i \llbracket t \rrbracket
\end{aligned}$$

Produced terms use a combination of System F and dependent types, which is known as λ_{P2} [Barendregt, 1992] and is strongly normalizing. This PTS corresponds to second-order (intuitionistic) predicate logic and is part of Barendregt's *lambda cube*; λ_{P2} does not admit types depending on types (that is, type-level functions), but admits all other forms of abstraction in the lambda cube (terms on terms like $\lambda_\to$, terms on types like System F, and types on terms like LF/λ_P).

Finally, we sketch a transformation producing proofs that differentiation is correct for System F.

$$\begin{aligned}
&\mathcal{DP} \llbracket \varepsilon \rrbracket = \varepsilon \\
&\mathcal{DP} \llbracket \Gamma, x : \tau \rrbracket = \mathcal{DP} \llbracket \Gamma \rrbracket, x_1 : \mathcal{S}_1 \llbracket \tau \rrbracket, x_2 : \mathcal{S}_2 \llbracket \tau \rrbracket, \\
&\quad dx : (x_1, x_2) \in \Delta_2 \llbracket \tau \rrbracket, dxx : (x_1, x_2, dx) \in \Delta \mathcal{V} \llbracket \tau \rrbracket \\
&\mathcal{DP} \llbracket \Gamma, \alpha : \star \rrbracket = \mathcal{DP} \llbracket \Gamma \rrbracket,
\end{aligned}$$

$$
\begin{aligned}
&\alpha_1 : \star, \alpha_2 : \star, \Delta_\alpha : \alpha_1 \to \alpha_2 \to \star \\
&\mathcal{R}^\alpha : \Pi(x_1 : \alpha_1)\ (x_2 : \alpha_2)\ (dx : (x_1, x_2) \in \Delta_2 [\![\ \alpha\ ]\!]) \to \lceil \star \rceil \\
(f_1, f_2, df) \in \Delta\mathcal{V} [\![\ \forall \alpha.\ T\ ]\!] &= \\
&\Pi(\alpha_1 : \star)\ (\alpha_2 : \star)\ (\Delta_\alpha : \alpha_1 \to \alpha_2 \to \star) \\
&(\mathcal{R}^\alpha : \Pi(x_1 : \alpha_1)\ (x_2 : \alpha_2)\ (dx : (x_1, x_2) \in \Delta_2 [\![\ \alpha\ ]\!]) \to \lceil \star \rceil). \\
&\quad (f_1\ [\alpha_1], f_2\ [\alpha_2], df\ [\alpha_1]\ [\alpha_2]\ [\Delta_\alpha]) \in \Delta\mathcal{V} [\![\ T\ ]\!] \\
(f_1, f_2, df) \in \Delta\mathcal{V} [\![\ \sigma \to \tau\ ]\!] &= \\
&\Pi(x_1 : \mathcal{S}_1 [\![\ \sigma\ ]\!])\ (x_2 : \mathcal{S}_2 [\![\ \sigma\ ]\!])\ (dx : (x_1, x_2) \in \Delta_2 [\![\ \sigma\ ]\!])\ (dxx : (x_1, x_2, dx) \in \Delta\mathcal{V} [\![\ \sigma\ ]\!]). \\
&\quad (f_1\ x_1, f_2\ x_2, df\ x_1\ x_2\ dx) \in \Delta\mathcal{V} [\![\ \tau\ ]\!] \\
(x_1, x_2, dx) \in \Delta\mathcal{V} [\![\ \alpha\ ]\!] &= \mathcal{R}^\alpha\ x_1\ x_2\ dx \\
\mathcal{DP} [\![\ x\ ]\!] &= dxx \\
\mathcal{DP} [\![\ \lambda(x : \sigma) \to t\ ]\!] &= \\
&\lambda(x_1 : \mathcal{S}_1 [\![\ \sigma\ ]\!])\ (x_2 : \mathcal{S}_2 [\![\ \sigma\ ]\!])\ (dx : (x_1, x_2) \in \Delta_2 [\![\ \sigma\ ]\!]) \\
&\quad (dxx : (x_1, x_2, dx) \in \Delta\mathcal{V} [\![\ \sigma\ ]\!]) \to \\
&\quad \mathcal{DP} [\![\ t\ ]\!] \\
\mathcal{DP} [\![\ s\ t\ ]\!] &= \mathcal{DP} [\![\ s\ ]\!]\ \mathcal{S}_1 [\![\ t\ ]\!]\ \mathcal{S}_2 [\![\ t\ ]\!]\ \mathcal{D} [\![\ t\ ]\!]\ \mathcal{DP} [\![\ t\ ]\!] \\
\mathcal{DP} [\![\ \Lambda\alpha.\ t\ ]\!] &= \\
&\lambda(\alpha_1\ \alpha_2 : \star)\ (\Delta_\alpha : \alpha_1 \to \alpha_2 \to \star) \\
&\quad (\mathcal{R}^\alpha : \Pi(x_1 : \alpha_1)\ (x_2 : \alpha_2)\ (dx : (x_1, x_2) \in \Delta_2 [\![\ \alpha\ ]\!]) \to \lceil \star \rceil) \to \\
&\quad \mathcal{DP} [\![\ t\ ]\!] \\
\mathcal{DP} [\![\ t\ [\tau]\ ]\!] &= \mathcal{DP} [\![\ t\ ]\!]\ \mathcal{S}_1 [\![\ \tau\ ]\!]\ \mathcal{S}_2 [\![\ \tau\ ]\!]\ ((-, -) \in \Delta_2 [\![\ \tau\ ]\!])\ ((-, -, -) \in \Delta\mathcal{V} [\![\ \tau\ ]\!])
\end{aligned}
$$

Produced terms live in λ^2_{P2}, the logic produced by extending λ_{P2} following Bernardy and Lasson. A variant producing proofs for a non-dependently-typed differentiation (as suggested earlier) would produce proofs in λ^2_2, the logic produced by extending λ_2 following Bernardy and Lasson.

18.6 Related work

Dependently-typed differentiation for System F, as given, coincides with the parametricity transformation for System F given by Bernardy, Jansson, and Paterson [2010, Sec. 3.1]. But our application is fundamentally different: for known types, Bernardy et al. only consider identity relations, while we can consider non-identity relations as long as we assume that $\ominus$ is available for all types. What is more, Bernardy et al. do not consider changes, the update operator $\oplus$, or change structures across different types: changes are replaced by proofs of relatedness with no computational significance. Finally, non-dependently-typed differentiation (and its corectness proof) is novel here, as it makes limited sense in the context of parametricity, even though it is a small variant of Bernardy et al.'s parametricity transform.

18.7 Prototype implementation

We have written a prototype implementation of the above rules for a PTS presentation of System F, and verified it on a few representative examples of System F terms. We have built our implementation on top of an existing typechecker for PTSs.[4]

Since our implementation is defined in terms of a generic PTS, some of the rules presented above are unified and generalized in our implementation, suggesting the transformation might generalize

[4] https://github.com/Toxaris/pts/, by Tillmann Rendel, based on van Benthem Jutting et al. [1994]'s algorithm for PTS typechecking.

to arbitrary PTSs. In the absence of further evidence on the correctness of this generalization, we leave a detailed investigation as future work.

18.8 Chapter conclusion

In this chapter, we have sketched how to define and prove correct differentiation following Bernardy and Lasson [2011]'s and Bernardy et al. [2010]'s work on parametricity by code transformation. We give no formal correctness proof, but proofs appear mostly an extension of their methods. An important open point is Remark 18.2.1. We have implemented and tested differentiation in an existing mature PTS implementation, and verified it is type-correct on a few typical terms.

We leave further investigation as future work.

Chapter 19

Related work

Existing work on incremental computation can be divided into two groups: Static incrementalization and dynamic incrementalization. Static approaches analyze a program statically and generate an incremental version of it. Dynamic approaches create dynamic dependency graphs while the program runs and propagate changes along these graphs.

The trade-off between the two is that static approaches have the potential to be faster because no dependency tracking at runtime is needed, whereas dynamic approaches can support more expressive programming languages. ILC is a static approach, but compared to the other static approaches it supports more expressive languages.

In the remainder of this section, we analyze the relation to the most closely related prior works. Ramalingam and Reps [1993], Gupta and Mumick [1999] and Acar et al. [2006] discuss further related work. Other related work, more closely to cache-transfer style, is discussed in Sec. 17.6.

19.1 Dynamic approaches

One of the most advanced dynamic approach to incrementalization is self-adjusting computation, which has been applied to Standard ML and large subsets of C [Acar, 2009; Hammer et al., 2011]. In this approach, programs execute on the original input in an enhanced runtime environment that tracks the dependencies between values in a *dynamic dependence graph* [Acar et al., 2006]; intermediate results are memoized. Later, changes to the input propagate through dependency graphs from changed inputs to results, updating both intermediate and final results; this processing is often more efficient than recomputation.

However, creating dynamic dependence graphs imposes a large constant-factor overhead during runtime, ranging from 2 to 30 in reported experiments [Acar et al., 2009, 2010], and affecting the initial run of the program on its base input. Acar et al. [2010] show how to support high-level data types in the context of self-adjusting computation; however, the approach still requires expensive runtime bookkeeping during the initial run. Our approach, like other static ones, uses a standard runtime environment and has no overhead during base computation, but may be less efficient when processing changes. This pays off if the initial input is big compared to its changes.

Chen et al. [2011] have developed a static transformation for purely functional programs, but this transformation just provides a superior interface to use the runtime support with less boilerplate, and does not reduce this performance overhead. Hence, it is still a dynamic approach, unlike the transformation this work presents.

Another property of self-adjusting computation is that incrementalization is only efficient if the program has a suitable computation structure. For instance, a program folding the elements of a

bag with a left or right fold will not have efficient incremental behavior; instead, it's necessary that the fold be shaped like a balanced tree. In general, incremental computations become efficient only if they are *stable* [Acar, 2005]. Hence one may need to massage the program to make it efficient. Our methodology is different: Since we do not aim to incrementalize arbitrary programs written in standard programming languages, we can select primitives that have efficient derivatives and thereby require the programmer to use them.

Functional reactive programming [Elliott and Hudak, 1997] can also be seen as a dynamic approach to incremental computation; recent work by Maier and Odersky [2013] has focused on speeding up reactions to input changes by making them incremental on collections. Willis et al. [2008] use dynamic techniques to incrementalize JQL queries.

19.2 Static approaches

Static approaches analyze a program at compile-time and produce an incremental version that efficiently updates the output of the original program according to changing inputs.

Static approaches have the potential to be more efficient than dynamic approaches, because no bookkeeping at runtime is required. Also, the computed incremental versions can often be optimized using standard compiler techniques such as constant folding or inlining. However, none of them support first-class functions; some approaches have further restrictions.

Our aim is to apply static incrementalization to more expressive languages; in particular, ILC supports first-class functions and an open set of base types with associated primitive operations.

Sundaresh and Hudak [1991] propose to incrementalize programs using partial evaluation: given a partitioning of program inputs in parts that change and parts that stay constant, Sundaresh and Hudak partially evaluates a given program relative to the constant input parts, and combine the result with the changing inputs.

19.2.1 Finite differencing

Paige and Koenig [1982] present derivatives for a first-order language with a fixed set of primitives. Blakeley et al. [1986] apply these ideas to a class of relational queries. The database community extended this work to queries on relational data, such as in *algebraic differencing* [Gupta and Mumick, 1999], which inspired our work and terminology. However, most of this work does not apply to nested collections or algebraic data types, but only to relational (flat) data, and no previous approach handles first-class functions or programs resulting from defunctionalization or closure conversion. Incremental support is typically designed monolithically for a whole language, rather than piecewise. Improving on algebraic differencing, DBToaster (Koch [2010]; Koch et al. [2014]) *guarantees* asymptotic speedups with a compositional query transformation and delivers huge speedup in realistic benchmarks, though still for a first-order database language.

More general (non-relational) data types are considered in the work by Gluche et al. [1997]; our support for bags and the use of groups is inspired by their work, but their architecture is still rather restrictive: they lack support for function changes and restrict incrementalization to self-maintainable views, without hinting at a possible solution.

It seems possible to transform higher-order functional programs to database queries, using a variety of approaches [Grust et al., 2009; Cheney et al., 2013], some of which support first-class functions via closure conversion [Grust and Ulrich, 2013; Grust et al., 2013], and incrementalize the resulting programs using standard database technology. Such a solution would inherit limitations of database incrementalization approaches: in particular, it appears that database incrementalization approaches such as DBToaster can handle the insertion and removal of entire table rows, not of smaller changes. Nevertheless, such an alternative approach might be worth investigating.

Unlike later approaches to higher-order differentiation, we do not restrict our base types to groups unlike Koch et al. [2016], and transformed programs we produce do not require further runtime transformation unlike Koch et al. [2016] and Huesca [2015], as we discuss further next.

19.2.2 λ–diff and partial differentials

In work concurrent with Cai et al. [2014], Huesca [2015] introduces an alternative formalism, called λ–diff, for incremental computation by program transformation. While λ–diff has some appealing features, it currently appears to require program transformations at runtime. Other systems appear to share this feature [Koch et al., 2016]. Hence, this section discusses the reason in some detail.

Instead of differentiating a term t relative to all inputs (free variables and function arguments) via $\mathcal{D}[\![t]\!]$, like ILC, λ–diff differentiates terms relative to one input variable, and writes ${}^{\partial t}/_{\partial x,\, d_x}$ for the result of differentiating t relative to x, a term that computes the change in t when the value for x is updated by change d_x. The formalism also uses pointwise function changes, similarly to what we discussed in Sec. 15.3.

Unfortunately, it is not known how to define such a transformation for a λ-calculus with a standard semantics, and the needed semantics appear to require runtime term transformations, which are usually considered problematic when implementing functional languages. In particular, it appears necessary to introduce a new term constructor $D\ t$, which evaluates t to a function value $\lambda y \to u$, and then evaluates to $\lambda(y, d_y) \to {}^{\partial t}/_{\partial y,\, d_y}$, which differentiates t at runtime relative to its head variable y. As an indirect consequence, if the program under incrementalization contains function term $\Gamma \vdash t : \sigma \to \tau$, where Γ contains n free variables, it can be necessary in the worst-case to differentiate t over any subset of the n free variables in Γ. There are 2^n such subsets. To perform all term transformations before runtime, it seems hence necessary in the worst case to precompute 2^n partial derivatives for each function term t, which appears unfeasible. On the other hand, it is not clear how often this worst-case is realized, or how big n grows in typical programs, or if it is simply feasible to perform differentiation at runtime, similarly to JIT compilation. Overall, an efficient implementation of λ–diff remains an open problem. It appears also Koch et al. [2016] suffer similar problems, but a few details appear simpler since they restrict focus to functions over groups.

To see why λ–diff need introduce $D\ t$, consider differentiating ${}^{\partial s\ t}/_{\partial x,\, d_x}$, that is, the change d of $s\ t$ when xx is updated by change d_x. Change d depends (a) on the change of t when x is updated by d_x, that is ${}^{\partial t}/_{\partial x,\, d_x}$; (b) on how s changes when its input t is updated by ${}^{\partial t}/_{\partial x,\, d_x}$; to express this change, λ–diff expresses this via $(D\ s)\ t\ {}^{\partial t}/_{\partial x,\, d_x}$; (c) on the change of s (applied to the updated t) when x is updated by d_x, that is ${}^{\partial t}/_{\partial x,\, d_x}$. To compute component (b), λ–diff writes $D\ s$ to differentiate s not relative to x, but relative to the still unknown head variable of s. If s evaluates to $\lambda y \to u$, then y is the head variable of s, and $D\ s$ differentiates u relative to y and evaluates to $\lambda(y, d_y) \to {}^{\partial u}/_{\partial y,\, d_y}$.

Overall, the rule for differentiating application in λ-diff is

$$\frac{\partial s\ t}{\partial x, d_x} = (Ds)\left(t, \frac{\partial t}{\partial x, d_x}\right) \circledcirc \frac{\partial s}{\partial x, d_x}\left(t \oplus \frac{\partial t}{\partial x, d_x}\right).$$

This rule appears closely related to Eq. (15.1), hence we refer to its discussion for clarification.

On the other hand, differentiating a term relative to all its inputs introduces a different sort of overhead. For instance, it is much more efficient to differentiate $map\ f\ xs$ relative to xs than relative to f: if f undergoes a non-nil change df, $\mathcal{D}[\![map\ f\ xs]\!]$ must apply df to each elements in the updated input xs. Therefore, in our practical implementations $\mathcal{D}[\![map\ f\ xs]\!]$ tests whether df is nil and uses a more efficient implementation. In Chapter 16, we detect at compile time whether df is guaranteed to be nil. In Sec. 17.4.2, we instead detect at runtime whether df is nil. In both cases, authors of derivatives must implement this optimization by hand. Instead, λ–diff hints at a more general solution.

19.2.3 Static memoization

Liu's work [Liu, 2000] allows to incrementalize a first-order base program $f(x_{\text{old}})$ to compute $f(x_{\text{new}})$, knowing how x_{new} is related to x_{old}. To this end, they transform $f(x_{\text{new}})$ into an incremental program which reuses the intermediate results produced while computing $f(x_{\text{old}})$, the base program. To this end, (i) first the base program is transformed to save all its intermediate results, then (ii) the incremental program is transformed to reuse those intermediate results, and finally (iii) intermediate results which are not needed are pruned from the base program. However, to reuse intermediate results, the incremental program must often be rearranged, using some form of equational reasoning, into some equivalent program where partial results appear literally. For instance, if the base program f uses a left fold to sum the elements of a list of integers x_{old}, accessing them from the head onwards, and x_{new} prepends a new element h to the list, at no point does $f(x_{\text{new}})$ recompute the same results. But since addition is commutative on integers, we can rewrite $f(x_{\text{new}})$ as $f(x_{\text{old}}) + h$. The author's CACHET system will try to perform such rewritings automatically, but it is not guaranteed to succeed. Similarly, CACHET will try to synthesize any additional results which can be computed cheaply by the base program to help make the incremental program more efficient.

Since it is hard to fully automate such reasoning, we move equational reasoning to the plugin design phase. A plugin provides general-purpose higher-order primitives for which the plugin authors have devised efficient derivatives (by using equational reasoning in the design phase). Then, the differentiation algorithm computes incremental versions of user programs without requiring further user intervention. It would be useful to combine ILC with some form of static caching to make the computation of derivatives which are not self-maintainable more efficient. We plan to do so in future work.

Chapter 20

Conclusion and future work

In databases, standard finite differencing technology allows incrementalizing programs in a specific domain-specific first-order language of collection operations. Unlike other approaches, finite differencing transforms programs to programs, which can in turn be transformed further.

Differentiation (Chapter 10) transforms higher-order programs to incremental ones called *derivatives*, as long as one can provide incremental support for primitive types and operations.

Case studies in this thesis consider support for several such primitives. We first study a setting restricted to *self-maintainable* derivatives (Chapter 16). Then we study a more general setting where it is possible to remember inputs and intermediate results from one execution to the next, thanks to a transformation to *cache-transfer style* (Chapter 17). In all cases, we are able to deliver order-of-magnitude speedups on non-trivial examples; moreover, incrementalization produces programs in standard languages that are subject to further optimization and code transformation.

Correctness of incrementalization appeared initially surprising, so we devoted significant effort to formal correctness proofs, either on paper or in mechanized proof assistants. The original correctness proof of differentiation, using a set-theoretic denotational semantics [Cai et al., 2014], was a significant milestone, but since then we have simplified the proof to a logical relations proof that defines when a change is valid from a source to a destination, and proves that differentiation produces valid changes (Chapter 12). Valid changes witness the difference between sources and destinations; since changes can be *nil*, change validity arises as a generalization of the concept of *logical equivalence* and parametricity for a language (at least in terminating languages) (Chapter 18). Crucially, changes are not just witnesses: operation $\oplus$ takes a change and its source to the change destination. One can consider further operations, that give rise to an algebraic structure that we call *change structure* (Chapter 13).

Based on this simplified proof, in this thesis we generalize correctness to further languages using big-step operational semantics and (step-indexed) logical relations (Appendix C). Using operational semantics we reprove correctness for simply-typed λ-calculus, then add support for recursive functions (which would require domain-theoretic methods when using denotational semantics), and finally extend the proof to untyped λ-calculus. Building on a variant of this proof (Sec. 17.3.3), we show that conversion to cache-transfer-style also preserves correctness.

Based on a different formalism for logical equivalence and parametricity [Bernardy and Lasson, 2011], we sketch variants of the transformation and correctness proofs for simply-typed λ-calculus with type variables and then for full System F (Chapter 18).

Future work To extend differentiation to System F, we must consider changes where source and destination have different types. This generalization of changes makes change operations much

more flexible: it becomes possible to define a combinator language for change structures, and it appears possible to define nontrivial change structures for algebraic datatypes using this combinator language.

We conjecture such a combinator language will allow programmers to define correct change structures out of simple, reusable components.

Incrementalizing primitives correctly remains at present a significant challenge. We provide tools to support this task by formalizing equational reasoning, but it appears necessary to provide more tools to programmers, as done by Liu [2000]. Conceivably, it might be possible to build such a rewriting tool on top of the rewriting and automation support of theorem provers such as Coq.

Appendixes

Appendix A

Preliminaries: syntax and semantics of simply-typed λ-calculus

To discuss how we incrementalize programs and prove that our incrementalization technique gives correct results, we specify which foundation we use for our proofs and what object language we study throughout most of Part II.

We mechanize our correctness proof using Agda, hence we use Agda's underlying type theory as our foundation. We discuss what this means in Appendix A.1.

Our object language is a standard simply-typed λ-calculus (STLC) [Pierce, 2002, Ch. 9], parameterized over base types and constants. We term the set of base types and constants a *language plugin* (see Appendix A.2.5). In our examples we assume that the language plugins supports needed base types and constants. Later (e.g., in Chapter 12) we add further requirements to language plugins, to support incrementalization of the language features they add to our STLC. Rather than operational semantics we use a denotational semantics, which is however set-theoretic rather than domain-theoretic. Our object language and its semantics are summarized in Fig. A.1.

At this point, readers might want to skip to Chapter 10 right away, or focus on denotational semantics, and refer to this section as needed.

A.1 Our proof meta-language

In this section we describe the logic (or meta-language) used in our *mechanized* correctness proof.

First, as usual, we distinguish between "formalization" (that is, on-paper formalized proofs) and "mechanization" (that is, proofs encoded in the language of a proof assistant for computer-aided *mechanized* verification).

To prove the correctness of ILC, we provide a mechanized proof in Agda [Agda Development Team, 2013]. Agda implements intensional Martin-Löf type theory (from now on, simply type theory), so type theory is also the foundation of our proofs.

At times, we use conventional set-theoretic language to discuss our proofs, but the differences are only superficial. For instance, we might talk about a set of elements S and with elements such as $s \in S$, though we always mean that S is a metalanguage type, that s is a metalanguage value, and that $s : S$. Talking about sets avoids ambiguity between types of our meta-language and types of our object-language (that we discuss next in Appendix A.2).

Notation A.1.1
We'll let uppercase latin letters $A, B, C \ldots, V, U$ range over sets, never over types. □

We do not prove correctness of all our language plugins. However, in earlier work [Cai et al., 2014] we prove correctness for a language plugin supporting *bags*, a type of collection (described in Sec. 10.2). For that proof, we extend our logic by postulating a few standard axioms on the implementation of bags, to avoid proving correct an implementation of bags, or needing to account for different values representing the same bag (such different values typically arise when implementing bags as search trees).

A.1.1 Type theory versus set theory

Here we summarize a few features of type theory over set theory.

Type theory is dependently typed, so it generalizes function type $A \to B$ to *dependent* function type $(x : A) \to B$, where x can appear free in B. Such a type guarantees that if we apply a function $f : (x : A) \to B$ to an argument $a : A$, the result has type $B\,[x := a]$, that is B where x is substituted by a. At times, we will use dependent types in our presentation.

Moreover, by using type theory:

- We do not postulate the law of excluded middle; that is, our logic is constructive.
- Unlike set theory, type theory is proof-relevant: that is, proofs are first-class mathematical objects.
- Instead of subsets $\{x \in A \mid P(x)\}$, we must use Σ-types $\Sigma(x : A)P(x)$ which contain pairs of elements x and proofs they satisfy predicate P.
- In set theory, we can assume without further ado functional extensionality, that is, that functions that give equal results on all equal inputs are equal themselves. Intuitionistic type theory does not prove functional extensionality, so we need to add it as a postulate. In Agda, this postulate is known to be consistent [Hofmann, 1996], hence it is safe to assume[1].

To handle binding issues in our object language, our formalization uses typed de Bruijn indexes, because this techniques takes advantage of Agda's support for type refinement in pattern matching. On top of that, we implement a HOAS-like frontend, which we use for writing specific terms.

A.2 Simply-typed λ-calculus

We consider as object language a strongly-normalizing simply-typed λ-calculus (STLC). We choose STLC as it is the simplest language with first-class functions and types, while being a sufficient model of realistic total languages.[2] We recall the syntax and typing rules of STLC in Figs. A.1a and A.1b, together with metavariables we use. Language plugins define base types ι and constants c. Types can be base types ι or function types $\sigma \to \tau$. Terms can be constants c, variables x, function applications $t_1\ t_2$ or λ-abstractions $\lambda(x : \sigma) \to t$. To describe assumptions on variable types when typing terms, we define (typing) contexts Γ as being either empty ε, or as context extensions $\Gamma, x : \tau$, which extend context Γ by asserting variable x has type τ. Typing is defined through a judgment

[1] http://permalink.gmane.org/gmane.comp.lang.agda/2343
[2] To know why we restrict to total languages see Sec. 15.1.

$$
\begin{array}{rll}
\iota & ::= & \ldots \quad \text{(base types)} \\
\sigma, \tau & ::= & \iota \mid \tau \to \tau \quad \text{(types)} \\
\Gamma & ::= & \varepsilon \mid \Gamma, x : \tau \quad \text{(typing contexts)} \\
c & ::= & \ldots \quad \text{(constants)} \\
s, t & ::= & c \mid \lambda(x : \sigma) \to t \mid t\ t \mid x \quad \text{(terms)}
\end{array}
$$

(a) Syntax

$$
\frac{\vdash^{\mathcal{C}} c : \tau}{\Gamma \vdash c : \tau}\ \textsc{Const} \qquad \frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash x : \tau}\ \textsc{Lookup} \qquad \boxed{\Gamma \vdash t : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda(x : \sigma) \to t : \sigma \to \tau}\ \textsc{Lam}
$$

$$
\frac{\Gamma \vdash s : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}\ \textsc{App}
$$

(b) Typing

$$
\begin{array}{rl}
[\![\iota]\!] & = \ldots \\
[\![\sigma \to \tau]\!] & = [\![\sigma]\!] \to [\![\tau]\!]
\end{array}
$$

(c) Values.

$$
\begin{array}{rl}
[\![\varepsilon]\!] & = \{ \varnothing \} \\
[\![\Gamma, x : \tau]\!] & = \{ (\rho, x = v) \mid \rho \in [\![\Gamma]\!] \wedge v \in [\![\tau]\!] \}
\end{array}
$$

(d) Environments.

$$
\begin{array}{rl}
[\![c]\!]\ \rho & = [\![c]\!]^{\mathcal{C}} \\
[\![\lambda(x : \sigma) \to t]\!]\ \rho & = \lambda(v : [\![\sigma]\!]) \to [\![t]\!]\ (\rho, x = v) \\
[\![s\ t]\!]\ \rho & = ([\![s]\!]\ \rho)\ ([\![t]\!]\ \rho) \\
[\![x]\!]\ \rho & = \text{lookup } x \text{ in } \rho
\end{array}
$$

(e) Denotational semantics.

Figure A.1: Standard definitions for the simply-typed lambda calculus.

$\Gamma \vdash t : \tau$, stating that term t under context Γ has type τ.[3] For a proper introduction to STLC we refer the reader to Pierce [2002, Ch. 9]. We will assume significant familiarity with it.

An extensible syntax of types In fact, the definition of base types can be mutually recursive with the definition of types. So a language plugin might add as base types, for instance, collections of elements of type τ, products and sums of type σ and type τ, and so on. However, this mutual recursion must satisfy a few technical restrictions to avoid introducing subtle inconsistencies, and Agda cannot enforce these restrictions across modules. Hence, if we define language plugins as separate modules in our mechanization, we need to verify *by hand* that such restrictions are satisfied (which they are). See Appendix B.1 for the gory details.

Notation A.2.1
We typically omit type annotations on λ-abstractions, that is we write $\lambda x \to t$ rather than $\lambda(x : \sigma) \to t$. Such type annotations can often be inferred from context (or type inference). Nevertheless,

[3] We only formalize typed terms, not untyped ones, so that each term has a unique type. That is, in the relevant jargon, we use *Church-style* typing as opposed to *Curry-style* typing. Alternatively, we use an intrinsically-typed term representation. In fact, arguably we mechanize at once both well-typed terms and their typing derivations. This is even more clear in our mechanization; see discussion in Appendix A.2.4.

whenever we discuss terms of shape $\lambda x \to t$, we're in fact discussing $\lambda(x : \sigma) \to t$ for some arbitrary σ. We write $\lambda x \to t$ instead of $\lambda x.\ t$, for consistency with the notation we use later for Haskell programs.

We often omit ε from typing contexts with some assumptions. For instance we write $x : \tau_1, y : \tau_2$ instead of $\varepsilon, x : \tau_1, y : \tau_2$.

We overload symbols (often without warning) when they can be disambiguated from context, especially when we can teach modern programming languages to disambiguate such overloadings. For instance, we reuse $\to$ for lambda abstractions $\lambda x \to t$, function spaces $A \to B$, and function types $\sigma \to \tau$, even though the first is the separator. □

Extensions In our examples, we will use some unproblematic syntactic sugar over STLC, including let expressions, global definitions, type inference, and we will use a Haskell-like concrete syntax. In particular, when giving type signatures or type annotations in Haskell snippets, we will use :: to separate terms or variables from their types, rather than : as in λ-calculus. To avoid confusion, we never use : to denote the constructor for Haskell lists.

At times, our concrete examples will use Hindley-Milner (prenex) polymorphism, but this is also not such a significant extension. A top-level definition using prenex polymorphism, that is of type $\forall\alpha.\ \tau$ (where α is free in τ), can be taken as sugar for a metalevel family of object-level programs, indexed by type argument τ_1 of definitions of type $\tau\ [\alpha := \tau_1]$. We use this trick without explicit mention in our first implementation of incrementalization in Scala [Cai et al., 2014].

A.2.1 Denotational semantics for STLC

To prove that incrementalization preserves the semantics of our object-language programs, we define a semantics for STLC. We use a naive set-theoretic denotational semantics: Since STLC is strongly normalizing [Pierce, 2002, Ch. 12], its semantics need not handle partiality. Hence, we can use denotational semantics but eschew using domain theory, and simply use sets from the metalanguage (see Appendix A.1). Likewise, we can use normal functions as domains for function types.

We first associate, to every type τ, a set of values $[\![\ \tau\ ]\!]$, so that terms of a type τ evaluate to values in $[\![\ \tau\ ]\!]$. We call set $[\![\ \tau\ ]\!]$ a *domain*. Domains associated to types τ depend on domain associated to base types ι, that must be specified by language plugins (Plugin Requirement A.2.10).

Definition A.2.2 (Domains and values)
The domain $[\![\ \tau\ ]\!]$ of a type τ is defined as in Fig. A.1c. A value is a member of a domain. □

We let metavariables $u, v, \ldots, a, b, \ldots$ range over members of domains; we tend to use v for generic values and a for values we use as a function argument. We also let metavariable $f, g, \ldots$ range over values in the domain for a function type. At times we might also use metavariables $f, g, \ldots$ to range over *terms* of function types; the context will clarify what is intended.

Given this domain construction, we can now define a denotational semantics for terms. The plugin has to provide the evaluation function for constants. In general, the evaluation function $[\![\ t\ ]\!]$ computes the value of a well-typed term t given the values of all free variables in t. The values of the free variables are provided in an environment.

Definition A.2.3 (Environments)
An environment ρ assigns values to the names of free variables.

$$\rho ::= \varepsilon \mid \rho, x = v$$

We write $[\![\ \Gamma\ ]\!]$ for the set of environments that assign values to the names bound in Γ (see Fig. A.1d). □

Notation We often omit $\varnothing$ from environments with some assignments. For instance we write $x = v_1, y = v_2$ instead of $\varnothing, x = v_1, y = v_2$.

Definition A.2.4 (Evaluation)
Given $\Gamma \vdash t : \tau$, the meaning of t is defined by the function $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![\tau]\!]$ in Fig. A.1e.□

This is a standard denotational semantics of the simply-typed λ-calculus.

For each constant $c : \tau$, the plugin provides $[\![c]\!]^{\mathcal{C}} : [\![\tau]\!]$, the semantics of c (by Plugin Requirement A.2.11); since constants don't contain free variables, $[\![c]\!]^{\mathcal{C}}$ does not depend on an environment.

We define a program equivalence across terms of the same type $t_1 \cong t_2$ to mean $[\![t_1]\!] = [\![t_2]\!]$.

Definition A.2.5 (Denotational equivalence)
We say that two terms $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$ are denotationally equal, and write $\Gamma \vDash t_1 \cong t_2 : \tau$ (or sometimes $t_1 \cong t_2$), if for all environments $\rho : [\![\Gamma]\!]$ we have that $[\![t_1]\!]\ \rho = [\![t_2]\!]\ \rho$. □

Remark A.2.6
Beware that denotational equivalence cannot always be strengthened by dropping unused variables: that is, $\Gamma, x : \sigma \vDash t_1 \cong t_2 : \tau$ does not imply $\Gamma \vDash t_1 \cong t_2 : \tau$, even if x does not occur free in either t_1 or t_2. Counterexamples rely on σ being an empty type. For instance, we cannot weaken $x : \mathbf{0}_\tau \vDash 0 \cong 1 : \mathbb{Z}$ (where $\mathbf{0}_\tau$ is an empty type): this equality is only true vacuously, because there exists no environment for context $x : \mathbf{0}_\tau$. □

A.2.2 Weakening

While we don't discuss our formalization of variables in full, in this subsection we discuss briefly weakening on STLC terms and state as a lemma that weakening preserves meaning. This lemma is needed in a key proof, the one of Theorem 12.2.2.

As usual, if a term t is well-typed in a given context Γ_1, and context Γ_2 extends Γ_1 (which we write as $\Gamma_1 \preceq \Gamma_2$), then t is also well-typed in Γ_2.

Lemma A.2.7 (Weakening is admissible)
The following typing rule is admissible:

$$\frac{\Gamma_1 \vdash t : \tau \qquad \Gamma_1 \preceq \Gamma_2}{\Gamma_2 \vdash t : \tau}\ \textsc{Weaken}$$

□

Weakening also preserves semantics. If a term t is typed in context Γ_1, evaluating it requires an environment matching Γ_1. So if we weaken t to a bigger context Γ_2, evaluation requires an extended environment matching Γ_2, and is going to produce the same result.

Lemma A.2.8 (Weakening preserves meaning)
Take $\Gamma_1 \vdash t : \tau$ and $\rho_1 : [\![\Gamma_1]\!]$. If $\Gamma_1 \preceq \Gamma_2$ and $\rho_2 : [\![\Gamma_2]\!]$ extends ρ_1, then we have that

$$[\![t]\!]\ \rho_1 = [\![t]\!]\ \rho_2.$$

□

Mechanize these statements and their proofs requires some care. We have a meta-level type $\mathit{Term}\ \Gamma\ \tau$ of object terms having type τ in context Γ. Evaluation has type $[\![-]\!] : \mathit{Term}\ \Gamma\ \tau \to [\![\Gamma]\!] \to [\![\tau]\!]$, so $[\![t]\!]\ \rho_1 = [\![t]\!]\ \rho_2$ is not directly ill-typed. To remedy this, we define formally the subcontext relation $\Gamma_1 \preceq \Gamma_2$, and an explicit operation that weakens a term in context Γ_1 to a corresponding term in bigger context Γ_2, $\mathit{weaken} : \Gamma_1 \preceq \Gamma_2 \to \mathit{Term}\ \Gamma_1\ \tau \to \mathit{Term}\ \Gamma_2\ \tau$. We define the subcontext relation $\Gamma_1 \preceq \Gamma_2$ as a judgment using *order preserving embeddings*.[4] We refer to our mechanized proof for details, including auxiliary definitions and relevant lemmas.

[4] As mentioned by James Chapman at https://lists.chalmers.se/pipermail/agda/2011/003423.html, who attributes them to Conor McBride.

A.2.3 Substitution

Some facts can be presented using (capture-avoiding) substitution rather than environments, and we do so at some points, so let us fix notation. We write $t\ [x := s]$ for the result of substituting variable x in term t by term s.

We have mostly avoided mechanizing proofs about substitution, but we have mechanized substitution following Keller and Altenkirch [2010] and proved the following substitution lemma:

Lemma A.2.9 (Substitution lemma)
For any term $\Gamma \vdash t : \tau$, variable $x : \sigma$ bound in Γ, we write $\Gamma - x$ for the result of removing variable x from Γ (as defined by Keller and Altenkirch). Take term $\Gamma - x \vdash s : \sigma$, and environment $\rho : [\![\, \Gamma - x \,]\!]$. Then, we have that substitution and evaluation commute as follows:

$$[\![\, t\ [x := s] \,]\!]\ \rho = [\![\, t \,]\!]\ (\rho, x = [\![\, s \,]\!]\ \rho).$$

□

A.2.4 Discussion: Our mechanization and semantic style

To formalize meaning of our programs, we use denotational semantics while nowadays most prefer operational semantics, in particular small-step. Hence, we next justify our choice and discuss related work.

We expect we could use other semantics techniques, such as big-step or small-step semantics. But at least for such a simple object language, working with denotational semantics as we use it is easier than other approaches in a proof assistant, especially in Agda.

- Our semantics $[\![\, - \,]\!]$ is a function and not a relation, like in small-step or big-step semantics.
- It is clear to Agda that our semantics is a total function, since it is structurally recursive.
- Agda can normalize $[\![\, - \,]\!]$ on partially-known terms when normalizing goals.
- The meanings of our programs are well-behaved Agda functions, not syntax, so we know "what they mean" and need not prove any lemmas about it. We need not prove, say, that evaluation is deterministic.

In Agda, the domains for our denotational semantics are simply Agda types, and semantic values are Agda values — in other words, we give a denotational semantics in terms of type theory. Using denotational semantics allows us to state the specification of differentiation directly in the semantic domain, and take advantage of Agda's support for equational reasoning for proving equalities between Agda functions.

Related work Our variant is used for instance by McBride [2010], who attribute it to Augustsson and Carlsson [1999] and Altenkirch and Reus [1999]. In particular, Altenkirch and Reus [1999] already define our type *Term* $\Gamma\ \tau$ of simply-typed λ-terms t, well-typed with type τ in context Γ, while Augustsson and Carlsson [1999] define semantic domains by induction over types. Benton et al. [2012] and Allais et al. [2017] also discuss this approach to formalizing λ terms, and discuss how to best prove various lemmas needed to reason, for instance, about substitution.

More in general, similar approaches are becoming more common when using proof assistants. Our denotational semantics could be otherwise called a *definitional interpreter* (which is in particular compositional), and mechanized formalizations using a variety of definitional interpreters are nowadays often advocated, either using denotational semantics [Chlipala, 2008], or using *functional* big-step semantics. Functional semantics are so convenient that their use has been advocated even for languages that are *not* strongly normalizing [Owens et al., 2016; Amin and Rompf, 2017], even at the cost of dealing with step-indexes.

A.2.5 Language plugins

Our object language is parameterized by *language plugins* (or just plugins) that encapsulate its domain-specific aspects.

In our examples, our language plugin will typically support integers and primitive operations on them. However, our plugin will also support various sorts *collections* and base operations on them. Our first example of collection will be *bags*. Bags are unordered collections (like sets) where elements are allowed to appear more than once (unlike in sets), and they are also called multisets.

Our formalization is parameterized over one language plugin providing all base types and primitives. In fact, we expect a language plugin to be composed out of multiple language plugins merged together [Erdweg et al., 2012]. Our mechanization is mostly parameterized over language plugins, but see Appendix B.1 for discussion of a few limitations.

The sets of base types and primitive constants, as well as the types for primitive constants, are on purpose left unspecified and only defined by plugins — they are *extensions points*. We write some extension points using ellipses ("..."), and other ones by creating names, which typically use ^{C} as a superscript.

A plugin defines a set of base types ι, primitives c and their denotational semantics $[\![c]\!]^C$. As usual, we require that $[\![c]\!]^C : [\![\tau]\!]$ whenever $c : \tau$.

Summary To sum up the discussion of plugins, we collect formally the plugin requirements we have mentioned in this chapter.

Plugin Requirement A.2.10 (Base types)
There is a set of base types ι, and for each there is a domain $[\![\iota]\!]$. □

Plugin Requirement A.2.11 (Constants)
There is a set of constants c. To each constant is associated a type τ, such that the constant has that type, that is $\vdash^C c : \tau$, and the constants' semantics matches that type, that is $[\![c]\!]^C : [\![\tau]\!]$. □

After discussing the metalanguage of our proofs, the object language we study, and its semantics, we begin discussing incrementalization in next chapter.

Appendix B

More on our formalization

B.1 Mechanizing plugins modularly and limitations

Next, we discuss our mechanization of language plugins in Agda, and its limitations. For the concerned reader, we can say these limitations affect essentially how modular our proofs are, and not their cogency.

In essence, it's not possible to express in Agda the correct interface for language plugins, so some parts of language plugins can't be modularized as desirable. Alternatively, we can mechanize any fixed language plugin together with the main formalization, which is not truly modular, or mechanize a core formulation parameterized on a language plugin, which that runs into a few limitations, or encode plugins so they can be modularized and deal with encoding overhead.

This section requires some Agda knowledge not provided here, but we hope that readers familiar with both Haskell and Coq will be able to follow along.

Our mechanization is divided into multiple Agda modules. Most modules have definitions that depend on language plugins, directly or indirectly. Those take definitions from language plugins as *module parameters*.

For instance, STLC object types are formalized through Agda type *Type*, defined in module *Parametric.Syntax.Type*. The latter is parameterized over *Base*, the type of base types.

For instance, *Base* can support a base type of integers, and a base type of bags of elements of type ι (where ι : *Base*). Simplifying a few details, our definition is equivalent to the following Agda code:

```
module Parametric.Syntax.Type (Base : Set) where
  data Type : Set where
    base : (ι : Base) → Type
    _⇒_ : (σ : Type) → (τ : Type) → Type
  -- Elsewhere, in plugin:
data Base : Set where
  baseInt : Base
  baseBag : (ι : Base) → Base
  -- ...
```

But with these definitions, we only have bags of elements of base type. If ι is a base type, *base* (*baseBag* ι) is the type of bags with elements of type *base* ι. Hence, we have bags of elements of base type. But we don't have a way to encode *Bag* τ if τ is an arbitrary non-base type, such as

base baseInt ⇒ *base baseInt* (the encoding of object type $\mathbb{Z} \to \mathbb{Z}$). Can we do better? If we ignore modularization, we can define types through the following mutually recursive datatypes:

```
mutual
  data Type : Set where
    base  :  (ι : Base Type) → Type
    _⇒_ :  (σ : Type) → (τ : Type) → Type
  data Base : Set where
    baseInt  :  Base
    baseBag :  (ι : Type) → Base
```

So far so good, but these types have to be defined together. We can go a step further by defining:

```
mutual
  data Type : Set where
    base  :  (ι : Base Type) → Type
    _⇒_ :  (σ : Type) → (τ : Type) → Type
  data Base (Type : Set) : Set where
    baseInt  :  Base Type
    baseBag :  (ι : Type) → Base Type
```

Here, *Base* takes the type of object types as a parameter, and *Type* uses *Base Type* to tie the recursion. This definition still works, but only as long as *Base* and *Type* are defined together.

If we try to separate the definitions of *Base* and *Type* into different modules, we run into trouble.

```
module Parametric.Syntax.Type (Base : Set → Set) where
  data Type : Set where
    base  :  (ι : Base Type) → Type
    _⇒_ :  (σ : Type) → (τ : Type) → Type
  -- Elsewhere, in plugin:
data Base (Type : Set) : Set where
  baseInt  :  Base Type
  baseBag :  (ι : Type) → Base Type
```

Here, *Type* is defined for an arbitrary function on types *Base* : *Set* → *Set*. However, this definition is rejected by Agda's *positivity checker*. Like Coq, Agda forbids defining datatypes that are not strictly positive, as they can introduce inconsistencies.

The above definition of *Type* is *not* strictly positive, because we could pass to it as argument $Base = \lambda\tau \to (\tau \to \tau)$ so that *Base Type* = *Type* → *Type*, making *Type* occur in a negative position. However, the actual uses of *Base* we are interested in are fine. The problem is that we cannot inform the positivity checker that *Base* is supposed to be a strictly positive type function, because Agda doesn't supply the needed expressivity.

This problem is well-known. It could be solved if Agda function spaces supported positivity annotations,[1] or by encoding a universe of strictly-positive type constructors. This encoding is not fully transparent and adds hard-to-hide noise to development [Schwaab and Siek, 2013].[2] Few alternatives remain:

- We can forbid types from occurring in base types, as we did in our original paper [Cai et al., 2014]. There we did not discuss at all a recursive definition of base types.

[1] As discussed in https://github.com/agda/agda/issues/570 and https://github.com/agda/agda/issues/2411.

[2] Using pattern synonyms and `DISPLAY` pragmas might successfully hide this noise.

- We can use the modular mechanization, disable positivity checking and risk introducing inconsistencies. We tried this successfully in a branch of that formalization. We believe we did not introduce inconsistencies in that branch but have no hard evidence.
- We can simply combine the core formalization with a sample plugin. This is not truly modular because the core modularization can only be reused by copy-and-paste. Moreover, in dependently-typed languages the content of a definition can affect whether later definitions typecheck, so alternative plugins using the same interface might not typecheck.[3]

Sadly, giving up on modularity altogether appears the more conventional choice. Either way, as we claimed at the outset, these modularity concerns only limit the modularity of the mechanized proofs, not their cogency.

[3]Indeed, if *Base* were not strictly positive, the application *Type Base* would be ill-typed as it would fail positivity checking, even though $Base : Set \to Set$ does not require *Base* to be strictly positive.

Appendix C

(Un)typed ILC, operationally

In Chapter 12 we have proved ILC correct for a simply-typed λ-calculus. What about other languages, with more expressive type systems or no type system at all?

In this chapter, we prove that ILC is still correct in untyped call-by-value (CBV) λ-calculus. We do so without using denotational semantics, but using only an environment-based big-step operational semantics and *step-indexed logical relations*. The formal development in this chapter stands alone from the rest of the thesis, though we do not repeat ideas present elsewhere.

We prove ILC correct using, in increasing order of complexity,

1. STLC and standard syntactic logical relations;
2. STLC and step-indexed logical relations;
3. an untyped λ-calculus and step-indexed logical relations.

We have fully mechanized the second proof in Agda[1], and done the others on paper. In all cases we prove the fundamental property for validity; we detail later which corollaries we prove in which case. The proof for untyped λ-calculus is the most interesting, but the others can serve as stepping stones. Yann Régis-Gianas, in collaboration with me, recently mechanized a similar proof for untyped λ-calculus in Coq, which appears in Sec. 17.3.[2]

Using operational semantics and step-indexed logical relations simplifies extending the proofs to more expressive languages, where denotational semantics or other forms of logical relations would require more sophistication, as argued by Ahmed [2006].

Proofs by (step-indexed) logical relations also promise to be scalable. All these proofs appear to be slight variants of proof techniques for logical program equivalence and parametricity, which are well-studied topics, suggesting the approach might scale to more expressive type systems. Hence, we believe these proofs clarify the relation with parametricity that has been noticed earlier [Atkey, 2015]. However, actually proving ILC correct for a polymorphic language (such as System F) is left as future work.

We also expect that from our logical relations, one might derive a logical *partial equivalence relation* among changes, similarly to Sec. 14.2, but we leave a development for future work.

Compared to earlier chapters, this one will be more technical and concise, because we already introduced the ideas behind both ILC and logical relation proofs.

[1] Source code available in this GitHub repo: https://github.com/inc-lc/ilc-agda.

[2] Mechanizing the proof for untyped λ-calculus is harder for purely technical reasons: mechanizing well-founded induction in Agda is harder than mechanizing structural induction.

Binding and environments On the technical side, we are able to mechanize our proof without needing any technical lemmas about binding or weakening, thanks to a number of choices we mention later.

Among other reasons, we avoid lemmas on binding because instead of substituting variables with arbitrary terms, we record mappings from variables to closed values via environments. We also used environments in Appendix A, but this time we use syntactic values rather than semantic ones. As a downside, on paper, using environments makes for more and bigger definitions, because we need to carry around both an environment and a term, instead of merging them into a single term via substitution, and because values are not a subset of terms but an entirely separate category. But on paper the extra work is straightforward, and in a mechanized setting it is simpler than substitution, especially in a setting with an intrinsically typed term representation (see Appendix A.2.4).

Background/related work Our development is inspired significantly by the use of step-indexed logical relations by Ahmed [2006] and Acar, Ahmed, and Blume [2008]. We refer to those works and to Ahmed's lectures at OPLSS 2013[3] for an introduction to (step-indexed) logical relations.

Intensional and extensional validity Until this point, change validity only specifies how function changes behave, that is, their *extension*. Using operational semantics, we can specify how valid function changes are concretely defined, that is, their *intension*. To distinguish the two concepts, we contrast extensional validity and intensional validity. Some extensionally valid changes are not intensionally valid, but such changes are never created by derivation. Defining intensional validity helps to understand function changes: function changes are produced from changes to values in environments or from functions being replaced altogether. Requiring intensional validity helps to implement change operations such as $\oplus$ more efficiently, by operating on environments. Later, in Appendix D, we use similar ideas to implement change operations on *defunctionalized* function changes: intensional validity helps put such efforts on more robust foundations, even though we do not account formally for defunctionalization but only for the use of closures in the semantics.

We use operational semantics to define extensional validity in Appendix C.3 (using plain logical relations) and Appendix C.4 (using step-indexed logical relations). We switch to intensional validity definition in Appendix C.5.

Non-termination and general recursion This proof implies correctness of ILC in the presence of general recursion, because untyped λ-calculus supports general recursion via fixpoint combinators. However, the proof only applies to terminating executions of base programs, like for earlier authors [Acar, Ahmed, and Blume, 2008]: we prove that if a function terminates against both a base input v_1 and an updated one v_2, its derivative terminates against the base input and a valid input change dv from v_1 to v_2.

We can also add a fixpoint construct to our *typed* λ-calculus and to our mechanization, without significant changes to our relations. However, a mechanical proof would require use of well-founded induction, which our current mechanization avoids. We return to this point in Appendix C.7.

While this support for general recursion is effective in some scenarios, other scenarios can still be incrementalized better using structural recursion. More efficient support for general recursion is a separate problem that we do not tackle here and leave for future work. We refer to discussion in Sec. 15.1.

Correctness statement Our final correctness theorem is a variant of Corollary 13.4.5, that we repeat for comparison:

[3] https://www.cs.uoregon.edu/research/summerschool/summer13/curriculum.html.

Corollary 13.4.5 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
If $\Gamma \vdash t : \tau$ and $d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : \Gamma$ then $[\![t]\!]\ \rho_1 \oplus [\![\mathcal{D}[\![t]\!]]\!]\ d\rho = [\![t]\!]\ \rho_2$. □

We present our final correctness theorem statement in Appendix C.5. We anticipate it here for illustration: the new statement is more explicit about evaluation, but otherwise broadly similar.

Corollary C.5.6 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
Take any term t that is well-typed ($\Gamma \vdash t : \tau$) and any suitable environments $\rho_1, d\rho, \rho_2$, intensionally valid at any step count ($\forall k.\, (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\Gamma]\!]$). Assume t terminates in both the old environment ρ_1 and the new environment ρ_2, evaluating to output values v_1 and v_2 ($\rho_1 \vdash t \Downarrow v_1$ and $\rho_2 \vdash t \Downarrow v_2$). Then $\mathcal{D}[\![t]\!]$ evaluates in environment ρ and change environment $d\rho$ to a change value dv ($\rho_1 \blacklozenge d\rho \vdash_\Delta t \Downarrow dv$), and dv is a valid change from v_1 to v_2, so that $v_1 \oplus dv = v_2$. □

Overall, in this chapter we present the following contributions:

- We give an alternative presentation of derivation, that can be mechanized without any binding-related lemmas, not even weakening-related ones, by introducing a separate syntax for change terms (Appendix C.1).
- We prove formally ILC correct for STLC using big-step semantics and logical relations (Appendix C.3).
- We show formally (with pen-and-paper proofs) that our semantics is equivalent to small-step semantics definitions (Appendix C.2).
- We introduce a formalized step-indexed variant of our definitions and proofs for simply-typed λ-calculus (Appendix C.3), which scales directly to definitions and proofs for *untyped* λ-calculus.
- For typed λ-calculus, we also mechanize our step-indexed proof.
- In addition to (extensional) validity, we introduce a concept of intensional validity, that captures formally that function changes arise from changing environments or functions being replaced altogether (Appendix C.5).

C.1 Formalization

To present the proofs, we first describe our formal model of CBV ANF λ-calculus. We define an untyped ANF language, called λ_A. We also define a simply-typed variant, called $\lambda_{A\to}$, by adding on top of λ_A a separate Curry-style type system.

In our mechanization of $\lambda_{A\to}$, however, we find it more convenient to define a Church-style type system (that is, a syntax that only describes typing derivations for well-typed terms) separately from the untyped language.

Terms resulting from differentiation satisfy additional invariants, and exploiting those invariants helps simplify the proof. Hence we define separate languages for change terms produced from differentiation, again in untyped (λ^{Δ}_{A}) and typed ($\lambda^{\Delta}_{A\to}$) variants.

The syntax is summarized in Fig. C.1, the type systems in Fig. C.2, and the semantics in Fig. C.3. The base languages are mostly standard, while the change languages are slightly more unusual.

$$
\begin{array}{lll}
p & ::= & \mathbf{succ} \mid \mathbf{add} \mid \ldots \\
m, n & ::= & \ldots \quad \text{-- numbers} \\
c & ::= & n \mid \ldots \\
w & ::= & x \mid \lambda x \to t \mid \langle w, w \rangle \mid c \\
s, t & ::= & w \mid w\ w \mid p\ w \mid \mathbf{let}\ x = t\ \mathbf{in}\ t \\
v & ::= & \rho\ [\lambda x \to t] \mid \langle v, v \rangle \mid c \\
\rho & ::= & x_1 := v_1, \ldots, x_n := v_n
\end{array}
$$

(a) Base terms (λ_A).

$$
\begin{array}{lll}
dc & ::= & 0 \\
dw & ::= & dx \mid \lambda x\ dx \to dt \mid \langle dw, dw \rangle \mid dc \\
ds, dt & ::= & dw \mid dw\ w\ dw \mid \mathbf{0}_p\ w\ dw \mid \\
 & & \quad \mathbf{let}\ x = t; dx = dt\ \mathbf{in}\ dt \\
dv & ::= & \rho \blacklozenge d\rho\ [\lambda x\ dx \to dt] \mid \langle dv, dv \rangle \mid dc \\
d\rho & ::= & dx_1 := dv_1, \ldots, dx_n := dv_n
\end{array}
$$

(b) Change terms (λ_A^Δ).

$$
\begin{aligned}
\mathcal{D}^C [\![\ n\ ]\!] &= 0 \\
\\
\mathcal{D} [\![\ x\ ]\!] &= dx \\
\mathcal{D} [\![\ \lambda x \to t\ ]\!] &= \lambda x\ dx \to \mathcal{D} [\![\ t\ ]\!] \\
\mathcal{D} [\![\ \langle w_a, w_b \rangle\ ]\!] &= \langle \mathcal{D} [\![\ w_a\ ]\!], \mathcal{D} [\![\ w_b\ ]\!] \rangle \\
\mathcal{D} [\![\ c\ ]\!] &= \mathcal{D}^C [\![\ c\ ]\!] \\
\\
\mathcal{D} \left[\!\!\left[\ w_f\ w_a\ \right]\!\!\right] &= \mathcal{D} \left[\!\!\left[\ w_f\ \right]\!\!\right]\ w_a\ \mathcal{D} [\![\ w_a\ ]\!] \\
\mathcal{D} [\![\ p\ w\ ]\!] &= \mathbf{0}_p\ w\ \mathcal{D} [\![\ w\ ]\!] \\
\mathcal{D} [\![\ \mathbf{let}\ x = s\ \mathbf{in}\ t\ ]\!] &= \mathbf{let}\ x = s; dx = \mathcal{D} [\![\ s\ ]\!]\ \mathbf{in}\ \mathcal{D} [\![\ t\ ]\!]
\end{aligned}
$$

(c) Differentiation.

$$
\begin{array}{lll}
\tau ::= \mathbb{N} \mid \tau_a \times \tau_b \mid \sigma \to \tau \\
\Delta\mathbb{N} & = & \mathbb{N} \\
\Delta(\tau_a \times \tau_b) & = & \Delta\tau_a \times \Delta\tau_b \\
\Delta(\sigma \to \tau) & = & \sigma \to \Delta\sigma \to \Delta\tau \\
\Delta\varepsilon & = & \varepsilon \\
\Delta(\Gamma, x : \tau) & = & \Delta\Gamma, dx : \Delta\tau
\end{array}
$$

(d) Types and contexts.

Figure C.1: ANF λ-calculus: λ_A and λ_A^Δ.

C.1.1 Types and contexts

We show the syntax of types, contexts and change types in Fig. C.1d. We introduce types for functions, binary products and naturals. Tuples can be encoded as usual through nested pairs. Change types are mostly like earlier, but this time we use naturals as change for naturals (hence, we cannot define a total $\ominus$ operation).

We modify the definition of change contexts and environment changes to *not* contain entries for base values: in this presentation we use separate environments for base variables and change variables. This choice avoids the need to define weakening lemmas.

C.1.2 Base syntax for λ_A

For convenience, we consider a λ-calculus in A-normal form. We do not parameterize this calculus over language plugins to reduce mechanization overhead, but we define separate syntactic categories for possible extension points.

We show the syntax of terms in Fig. C.1a.

Meta-variable v ranges over (closed) syntactic values, that is evaluation results. Values are numbers, pairs of values or closures. A closure is a pair of a function and an environment as usual. Environments ρ are finite maps from variables to syntactic values; in our mechanization using de Bruijn indexes, environments are in particular finite lists of syntactic values.

Meta-variable t ranges over arbitrary terms and w ranges over neutral forms. Neutral forms evaluate to values in zero steps, but unlike values they can be open: a neutral form is either a

$$\frac{}{\vdash_C n : \mathbb{N}}\ \text{T-Lit} \qquad \frac{}{\vdash_{\mathcal{P}} \mathbf{succ} : \mathbb{N} \to \mathbb{N}}\ \text{T-Succ} \qquad \frac{}{\vdash_{\mathcal{P}} \mathbf{add} : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}}\ \text{T-Add}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\vdash_C c : \tau}{\Gamma \vdash c : \tau}\ \text{T-Const} \qquad \boxed{\Gamma \vdash t : \tau} \qquad \frac{\Gamma \vdash w_f : \sigma \to \tau \qquad \Gamma \vdash w_a : \sigma}{\Gamma \vdash w_f\ w_a : \tau}\ \text{T-App}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x \to t : \sigma \to \tau}\ \text{T-Lam} \qquad \frac{\Gamma \vdash s : \sigma \qquad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathbf{let}\ x = s\ \mathbf{in}\ t : \tau}\ \text{T-Let} \qquad \frac{\Gamma \vdash w_a : \tau_a \qquad \Gamma \vdash w_b : \tau_b}{\Gamma \vdash \langle w_a, w_b \rangle : \tau_a \times \tau_b}\ \text{T-Pair}$$

$$\frac{\vdash_{\mathcal{P}} p : \sigma \to \tau \qquad \Gamma \vdash w : \sigma}{\Gamma \vdash p\ w : \tau}\ \text{T-Prim}$$

(a) $\lambda_{A\to}$ base typing.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_\Delta dx : \tau}\ \text{T-DVar} \qquad \frac{\vdash_C c : \Delta\tau}{\Gamma \vdash_\Delta c : \tau}\ \text{T-DConst} \qquad \boxed{\Gamma \vdash_\Delta dt : \tau}$$

$$\frac{\Gamma \vdash_\Delta dw_f : \sigma \to \tau \qquad \Gamma \vdash w_a : \sigma \qquad \Gamma \vdash_\Delta dw_a : \sigma}{\Gamma \vdash_\Delta dw_f\ w_a\ dw_a : \tau}\ \text{T-DApp}$$

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash_\Delta ds : \sigma \qquad \Gamma, x : \sigma \vdash_\Delta dt : \tau}{\Gamma \vdash_\Delta \mathbf{let}\ x = s; dx = ds\ \mathbf{in}\ dt : \tau}\ \text{T-DLet} \qquad \frac{\Gamma, x : \sigma \vdash_\Delta dt : \tau}{\Gamma \vdash_\Delta \lambda x\ dx \to dt : \sigma \to \tau}\ \text{T-DLam}$$

$$\frac{\Gamma \vdash_\Delta dw_a : \tau_a \qquad \Gamma \vdash_\Delta dw_b : \tau_b}{\Gamma \vdash_\Delta \langle dw_a, dw_b \rangle : \tau_a \times \tau_b}\ \text{T-DPair} \qquad \frac{\vdash_{\mathcal{P}} p : \sigma \to \tau \qquad \Gamma \vdash w : \sigma \qquad \Gamma \vdash_\Delta dw : \sigma}{\Gamma \vdash_\Delta \mathbf{0}_p\ w\ dw : \tau}\ \text{T-DPrim}$$

(b) $\lambda^{\Delta}_{A\to}$ change typing. Judgement $\Gamma \vdash_\Delta dt : \tau$ means that variables from both Γ and $\Delta\Gamma$ are in scope in dt, and the final type is in fact $\Delta\tau$.

Figure C.2: ANF λ-calculus, $\lambda_{A\to}$ and $\lambda^{\Delta}_{A\to}$ type system.

$$\frac{}{\rho \vdash p\ w \Downarrow_1 \mathcal{P}[\![\, p \,]\!]\,(\mathcal{V}[\![\, w \,]\!]\,\rho)}\ \text{E-Prim} \qquad \frac{}{\rho \vdash w \Downarrow_0 \mathcal{V}[\![\, w \,]\!]\,\rho}\ \text{E-Val} \qquad \boxed{\rho \vdash t \Downarrow_n v}$$

$$\frac{\rho \vdash s \Downarrow_{n_s} v_s \qquad (\rho, x := v_s) \vdash t \Downarrow_{n_t} v_t}{\rho \vdash \textbf{let}\ x = s\ \textbf{in}\ t \Downarrow_{1+n_s+n_t} v_t}\ \text{E-Let} \qquad \frac{\rho \vdash w_f \Downarrow_0 v_f \qquad \rho \vdash w_a \Downarrow_0 v_a \qquad \mathsf{app}\ v_f\ v_a \Downarrow_n v}{\rho \vdash w_f\ w_a \Downarrow_{1+n} v}\ \text{E-App}$$

$$\begin{aligned}
&\mathsf{app}\ (\rho'\ [\lambda x \to t])\ v_a = (\rho', x := v_a) \vdash t \\
&\mathcal{V}[\![\, x \,]\!]\,\rho && = \rho\ (x) \\
&\mathcal{V}[\![\, \lambda x \to t \,]\!]\,\rho && = \rho\ [\lambda x \to t] \\
&\mathcal{V}[\![\, \langle w_a, w_b \rangle \,]\!]\,\rho && = \langle \mathcal{V}[\![\, w_a \,]\!]\,\rho, \mathcal{V}[\![\, w_b \,]\!]\,\rho \rangle \\
&\mathcal{V}[\![\, c \,]\!]\,\rho && = c \\
&\mathcal{P}[\![\, \textbf{succ} \,]\!]\,n && = 1 + n \\
&\mathcal{P}[\![\, \textbf{add} \,]\!]\,(m, n) && = m + n
\end{aligned}$$

(a) Base semantics. Judgement $\rho \vdash t \Downarrow_n v$ says that $\rho \vdash t$, a pair of environment ρ and term t, evaluates to value v in n steps. Notation $\mathsf{app}\ v_f\ v_a \Downarrow_n v$ (used in rule E-App) is short for $\mathsf{app}\ v_f\ v_a = \rho \vdash t$ and $\rho \vdash t \Downarrow_n v$, that is, says that the pair $\rho \vdash t$ given by $\mathsf{app}\ v_f\ v_a$ evaluates to v in n steps.

$$\frac{}{\rho \blacklozenge d\rho \vdash_\Delta dw \Downarrow \mathcal{V}_\Delta[\![\, dw \,]\!]\,\rho\ d\rho}\ \text{E-DVal} \qquad \boxed{\rho \blacklozenge d\rho \vdash_\Delta dt \Downarrow dv}$$

$$\frac{}{\rho \blacklozenge d\rho \vdash_\Delta \mathbf{0}_p\ w\ dw \Downarrow \mathcal{P}_\Delta[\![\, p \,]\!]\,(\mathcal{V}[\![\, w \,]\!]\,\rho)\,(\mathcal{V}_\Delta[\![\, dw \,]\!]\,\rho\ d\rho)}\ \text{E-DPrim}$$

$$\frac{\rho \blacklozenge d\rho \vdash_\Delta dw_f \Downarrow dv_f \qquad \rho \vdash w_a \Downarrow v_a \qquad \rho \blacklozenge d\rho \vdash_\Delta dw_a \Downarrow dv_a \qquad \mathsf{dapp}\ dv_f\ v_a\ dv_a \Downarrow dv}{\rho \blacklozenge d\rho \vdash_\Delta dw_f\ w_a\ dw_a \Downarrow dv}\ \text{E-DApp}$$

$$\frac{\rho \vdash s \Downarrow v_s \qquad \rho \blacklozenge d\rho \vdash_\Delta ds \Downarrow dvs \qquad (\rho, x := v_s) \blacklozenge (d\rho; dx := dvs) \vdash_\Delta dt \Downarrow dvt}{\rho \blacklozenge d\rho \vdash_\Delta \textbf{let}\ x = s; dx = ds\ \textbf{in}\ dt \Downarrow dvt}\ \text{E-DLet}$$

$$\begin{aligned}
&\mathsf{dapp}\ (\rho' \blacklozenge d\rho'\ [\lambda x\ dx \to dt])\ v_a\ dv_a = (\rho', x := v_a) \blacklozenge (d\rho', dx := dv_a) \vdash_\Delta dt \\
&\mathcal{V}_\Delta[\![\, dx \,]\!]\,\rho\ d\rho && = d\rho\ (dx) \\
&\mathcal{V}_\Delta[\![\, \lambda x\ dx \to dt \,]\!]\,\rho\ d\rho && = \rho \blacklozenge d\rho\ [\lambda x\ dx \to dt] \\
&\mathcal{V}_\Delta[\![\, \langle dw_a, dw_b \rangle \,]\!]\,\rho\ d\rho && = \langle \mathcal{V}_\Delta[\![\, dw_a \,]\!]\,\rho\ d\rho, \mathcal{V}_\Delta[\![\, dw_b \,]\!]\,\rho\ d\rho \rangle \\
&\mathcal{V}_\Delta[\![\, c \,]\!]\,\rho\ d\rho && = c \\
&\mathcal{P}_\Delta[\![\, \textbf{succ} \,]\!]\,n\ dn && = dn \\
&\mathcal{P}_\Delta[\![\, \textbf{add} \,]\!]\,\langle m, n \rangle\ \langle dm, dn \rangle && = dm + dn
\end{aligned}$$

(b) Change semantics. Judgement $\rho \blacklozenge d\rho \vdash_\Delta t \Downarrow dv$ says that $\rho \blacklozenge d\rho \vdash_\Delta dt$, a triple of environment ρ, change environment $d\rho$ and change term t, evaluates to change value dv. Notation $\mathsf{dapp}\ dv_f\ v_a\ dv_a \Downarrow dv$ (used in rule E-DApp) is short for $\mathsf{dapp}\ dv_f\ v_a\ dv_a = \rho \blacklozenge d\rho \vdash_\Delta dt$ and $\rho \blacklozenge d\rho \vdash_\Delta t \Downarrow dv$, that is, says that the triple $\rho \blacklozenge d\rho \vdash_\Delta dt$ given by $\mathsf{dapp}\ dv_f\ v_a\ dv_a$ evalutes to dv.

Figure C.3: ANF λ-calculus (λ_A and λ_A^Δ), CBV semantics.

variable, a constant value c, a λ-abstraction or a pair of neutral forms.

A term is either a neutral form, an application of neutral forms, a let expression or an application of a primitive function p to a neutral form. Multi-argument primitives are encoded as primitives taking (nested) tuples of arguments. Here we use literal numbers as constants and +1 and addition as primitives (to illustrate different arities), but further primitives are possible.

Notation C.1.1
We use subscripts $_{ab}$ for pair components, $_{fa}$ for function and argument, and keep using $_{12}$ for old and new values. □

C.1.3 Change syntax for λ^{Δ}_{A}

Next, we consider a separate language for change terms, which can be transformed into the base language. This language supports directly the structure of change terms: base variables and change variables live in separate namespaces. As we show later, for the typed language those namespaces are represented by typing contexts Γ and $\Delta\Gamma$: that is, the typing context for change variables is always the change context for Γ.

We show the syntax of change terms in Fig. C.1b.

Change terms often take or bind two parameters at once, one for a base value and one for its change. Since a function change is applied to a base input and a change for it at once, the syntax for change term has a special binary application node $dw_f\ w_a\ dw_a$; otherwise, in ANF, such syntax must be encoded through separate applications via $\mathbf{let}\ df_a = dw_f\ w_a\ \mathbf{in}\ df_a\ dw_a$. In the same way, closure changes $\rho \bullet d\rho\ [\lambda x\ dx \to dt]$ bind two variables at once and close over separate environments for base and change variables. Various other changes in the same spirit simplify similar formalization and mechanization details.

In change terms, we write $\mathbf{0}_p$ as syntax for the derivative of p, evaluated as such by the semantics. Strictly speaking, differentiation *must* map primitives to standard terms, so that the resulting programs can be executed by a standard semantics; hence, we should replace $\mathbf{0}_p$ by a concrete implementation of the derivative of p. However, doing so in a new formalization yields little additional insight, and requires writing concrete derivatives of primitives as de Bruijn terms.

C.1.4 Differentiation

We show differentiation in Fig. C.1c. Differentiation maps constructs in the language of base terms one-to-one to constructs in the language of change terms.

C.1.5 Typing $\lambda_{A\to}$ and $\lambda^{\Delta}_{A\to}$

We define typing judgement for $\lambda_{A\to}$ base terms and for $\lambda^{\Delta}_{A\to}$ change terms. We show typing rules in Fig. C.2b.

Typing for base terms is mostly standard. We use judgements $\vdash_{\mathcal{P}} p$ and $\vdash_{C} c$ to specify typing of primitive functions and constant values. For change terms, one could expect a type system only proving judgements with shape $\Gamma, \Delta\Gamma \vdash dt : \Delta\tau$ (where $\Gamma, \Delta\Gamma$ stands for the concatenation of Γ and $\Delta\Gamma$). To simplify inversion on such judgements (especially in Agda), we write instead $\Gamma \vdash_{\Delta} dt : \tau$, so that one can verify the following derived typing rule for $\mathcal{D}[\![-]\!]$:

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash_{\Delta} \mathcal{D}[\![\, t \,]\!] : \tau}\ \text{T-Derive}$$

We also use mutually recursive typing judgment $\vDash v : \tau$ for values and $\vDash \rho : \Gamma$ for environments, and similarly $\vDash_\Delta dv : \tau$ for change values and $\vDash_\Delta d\rho : \Gamma$ for change environments. We only show the (unsurprising) rules for $\vDash v : \tau$ and omit the others. One could alternatively and equivalently define typing on syntactic values v by translating them to neutral forms $w = v*$ (using unsurprising definitions in Appendix C.2) and reusing typing on terms, but as usual we prefer to avoid substitution.

$$\frac{}{\vDash n : \mathbb{N}}\ \text{TV-Nat} \qquad \frac{\vDash v_a : \tau_a \qquad \vDash v_b : \tau_b}{\vDash \langle v_a, v_b \rangle : \tau_a \times \tau_b}\ \text{TV-Pair} \qquad \boxed{\vDash v : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau \qquad \vDash \rho : \Gamma}{\vDash \rho\ [\lambda x \to t] : \sigma \to \tau}\ \text{TV-Lam}$$

C.1.6 Semantics

We present our semantics for base terms in Fig. C.3a. Our semantics gives meaning to pairs of a environment ρ and term t, consistent with each other, that we write $\rho \vdash t$. By "consistent" we mean that ρ contains values for all free variables of t, and (in a typed language) values with compatible values (if $\Gamma \vdash t : \tau$ then $\vDash \rho : \Gamma$). Judgement $\rho \vdash t \Downarrow_n v$ says that $\rho \vdash t$ evaluates to value v in n steps. The definition is given via a CBV big-step semantics. Following Acar, Ahmed, and Blume [2008], we index our evaluation judgements via a step count, which counts in essence β-reduction steps; we use such step counts later, to define step-indexed logical relations. Since our semantics uses environments, β-reduction steps are implemented not via substitution but via environment extension, but the resulting step-counts are the same (Appendix C.2). Applying closure $v_f = \rho'\ [\lambda x \to t]$ to argument v_a produces environment-term pair $(\rho', x := v_a) \vdash t$, which we abbreviate as app $v_f\ v_a$. We'll reuse this syntax later to define logical relations.

In our mechanized formalization, we have additionally proved lemmas to ensure that this semantics is sound relative to our earlier denotational semantics (adapted for the ANF syntax).

Evaluation of primitives is delegated to function $\mathcal{P}[\![-]\!] -$. We show complete equations for the typed case; for the untyped case, we must turn $\mathcal{P}[\![-]\!] -$ and $\mathcal{P}_\Delta[\![-]\!] -$ into relations (or add explicit error results), but we omit the standard details (see also Appendix C.6). For simplicity, we assume evaluation of primitives takes one step. We conjecture higher-order primitives might need to be assigned different costs, but leave details for future work.

We can evaluate neutral forms w to syntactic values v using a simple evaluation function $\mathcal{V}[\![w]\!] \rho$, and use $\mathcal{P}[\![p]\!] v$ to evaluate primitives. When we need to omit indexes, we write $\rho \vdash t \Downarrow v$ to mean that for some n we have $\rho \vdash t \Downarrow_n v$.

We can also define an analogous non-indexed big-step semantics for change terms, and we present it in Fig. C.3b.

C.1.7 Type soundness

Evaluation preserves types in the expected way.

Lemma C.1.2 (Big-step preservation)

1. If $\Gamma \vdash t : \tau$, $\vDash \rho : \Gamma$ and $\rho \vdash t \Downarrow_n v$ then $\vDash v : \tau$.
2. If $\Gamma \vdash_\Delta dt : \tau$, $\vDash \rho : \Gamma$, $\vDash_\Delta d\rho : \Gamma$ and $\rho \blacklozenge d\rho \vdash_\Delta dt \Downarrow dv$ then $\vDash_\Delta dv : \tau$. □

Proof. By structural induction on evaluation judgements. In our intrinsically typed mechanization, this is actually part of the definition of values and big-step evaluation rules. □

To ensure that our semantics for $\lambda_{A\to}$ is complete for the typed language, instead of proving a small-step progress lemma or extending the semantics with errors, we just prove that all typed terms normalize in the standard way. As usual, this fails if we add fixpoints or for untyped terms. If we wanted to ensure type safety in such a case, we could switch to functional big-step semantics or definitional interpreters [Amin and Rompf, 2017; Owens et al., 2016].

Theorem C.1.3 (CBV normalization)
For any well-typed and closed term $\vdash t : \tau$, there exist a step count n and value v such that $\vdash t \Downarrow_n v$.□

Proof. A variant of standard proofs of normalization for STLC [Pierce, 2002, Ch. 12], adapted for big-step semantics rather than small-step semantics (similarly to Appendix C.3). We omit needed definitions and refer interested readers to our Agda mechanization. □

We haven't attempted to prove this lemma for arbitrary change terms (though we expect we could prove it by defining an erasure to the base language and relating evaluation results), but we prove it for the result of differentiating well-typed terms in Corollary C.3.2.

C.2 Validating our step-indexed semantics

In this section, we show how we ensure the step counts in our base semantics are set correctly, and how we can relate this environment-based semantics to more conventional semantics, based on substitution and/or small-step. We only consider the core calculus, without primitives, constants and pairs. Results from this section are not needed later and we have proved them formally on paper but not mechanized them, as our goal is to use environment-based big-step semantics in our mechanization.

To this end we relate our semantics first with a big-step semantics based on substitution (rather than environments) and then relating this alternative semantics to a small-step semantics. Results in this section are useful to understand better our semantics and as a design aide to modify it, but are not necessary to the proof, so we have not mechanized them.

As a consequence, we also conjecture that our logical relations and proofs could be adapted to small-step semantics, along the lines of Ahmed [2006]. We however do not find that necessary. While small-step semantics gives meaning to non-terminating programs, and that is important for type soundness proofs, it does not seem useful (or possible) to try to incrementalize them, or to ensure we do so correctly.

In proofs using step-indexed logical relations, the use of step-counts in definitions is often delicate and tricky to get right. But Acar, Ahmed, and Blume provide a robust recipe to ensure correct step-indexing in the semantics. To be able to prove the fundamental property of logical relations, we ensure step-counts agree with the ones induced by small-step semantics (which counts β-reductions). Such a lemma is not actually needed in other proofs, but only useful as a sanity check. We also attempted using the style of step-indexing used by Amin and Rompf [2017], but were unable to produce a proof. To the best of our knowledge all proofs using step-indexed logical relations, even with functional big-step semantics [Owens et al., 2016], use step-indexing that agrees with small-step semantics.

Unlike Acar, Ahmed, and Blume we use environments in our big-step semantics; this avoids the need to define substitution in our mechanized proof. Nevertheless, one can show the two semantics correspond to each other. Our environments ρ can be regarded as closed value substitutions, as long as we also substitute away environments in values. Formally, we write $\rho^*(t)$ for the "homomorphic

extension" of substitution ρ to terms, which produces other terms. If v is a value using environments, we write $w = v^*$ for the result of translating that value to not use environments; this operation produces a closed neutral form w. Operations $\rho^*(t)$ and v^* can be defined in a mutually recursive way:

$$\begin{aligned}
\rho^*(x) &= (\rho\ (x))^* \\
\rho^*(\lambda x \to t) &= \lambda x \to \rho^*(t) \\
\rho^*(w_1\ w_2) &= \rho^*(w_1)\ \rho^*(w_2) \\
\rho^*(\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2) &= \mathbf{let}\ x = \rho^*(t_1)\ \mathbf{in}\ \rho^*(t_2)
\end{aligned}$$

$$(\rho\ [\lambda x \to t])^* = \lambda x \to \rho^*(t)$$

If $\rho \vdash t \Downarrow_n v$ in our semantics, a standard induction over the derivation of $\rho \vdash t \Downarrow_n v$ shows that $\rho^*(t) \Downarrow_n v^*$ in a more conventional big-step semantics using substitution rather than environments (also following Acar, Ahmed, and Blume):

$$\frac{}{x \Downarrow_0 x}\ \textsc{Var'} \qquad \frac{}{\lambda x \to t \Downarrow_0 \lambda x \to t}\ \textsc{Lam'} \qquad \frac{t\ [x := w_2] \Downarrow_n w'}{(\lambda x \to t)\ w_2 \Downarrow_{1+n} w'}\ \textsc{App'}$$

$$\frac{t_1 \Downarrow_{n_1} w_1 \qquad t_2\ [x := w_1] \Downarrow_{n_2} w_2}{\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \Downarrow_{1+n_1+n_2} w_2}\ \textsc{Let'}$$

In this form, it is more apparent that the step indexes count steps of β-reduction or substitution.

It's also easy to see that this big-step semantics agrees with a standard small-step semantics $\mapsto$ (which we omit): if $t \Downarrow_n w$ then $t \mapsto^n w$. Overall, the two statements can be composed, so our original semantics agrees with small-step semantics: if $\rho \vdash t \Downarrow_n v$ then $\rho^*(t) \Downarrow_n v^*$ and finally $\rho^*(t) \mapsto^n v^*$. Hence, we can translate evaluation derivations using big-step semantics to derivations using small-step semantics *with the same step count.*

However, to state and prove the fundamental property we need not prove that our semantics is sound relative to some other semantics. We simply define the appropriate logical relation for validity and show it agrees with a suitable definition for $\oplus$.

Having defined our semantics, we proceed to define extensional validity.

C.3 Validity, syntactically ($\lambda_{A\to}$, $\lambda^{\Delta}_{A\to}$)

For our typed language $\lambda_{A\to}$, at least as long as we do not add fixpoint operators, we can define logical relations using big-step semantics but without using step-indexes. The resulting relations are well-founded only because they use structural recursion on types. We present in Fig. C.4 the needed definitions as a stepping stone to the definitions using step-indexed logical relations.

Following Ahmed [2006] and Acar, Ahmed, and Blume [2008], we encode extensional validity through two mutually recursive type-indexed families of ternary logical relations, $\mathcal{RV}\ [\![\ \tau\ ]\!]$ over closed values and $\mathcal{RC}\ [\![\ \tau\ ]\!]$ over terms (and environments).

These relations are analogous to notions we considered earlier and express similar informal notions.

- With denotational semantics, we write $dv \triangleright v_1 \hookrightarrow v_2 : \tau$ to say that change value $dv \in [\![\ \Delta\tau\ ]\!]$ is a valid change from v_1 to v_2 at type τ. With operational semantics instead we write $(v_1, dv, v_2) \in \mathcal{RV}\ [\![\ \tau\ ]\!]$, where v_1, dv and v_2 are now closed syntactic values.

- For terms, with denotational semantics we write $[\![\, dt \,]\!]\ d\rho \triangleright [\![\, t_1 \,]\!]\ \rho_1 \hookrightarrow [\![\, t_2 \,]\!]\ \rho_2 : \tau$ to say that dt is a valid change from t_1 and t_2, considering the respective environments. With operational semantics instead we write $(\rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\, \tau \,]\!]$.

Since we use Church typing and only mechanize typed terms, we must include in all cases appropriate typing assumptions.

Relation $\mathcal{RC}[\![\, \tau \,]\!]$ relates tuples of environments and computations, $\rho_1 \vdash t_1$, $\rho \bullet d\rho \vdash_\Delta dt$ and $\rho_2 \vdash t_2$: it holds if t_1 evaluates in environment ρ_1 to v_1, and t_2 evaluates in environment ρ_2 to v_2, then dt must evaluate in environments ρ and $d\rho$ to a change value dv, with v_1, dv, v_2 related by $\mathcal{RV}[\![\, \tau \,]\!]$. The environments themselves need not be related: this definition characterizes validity *extensionally*, that is, it can relate t_1, dt and t_2 that have unrelated implementations and unrelated environments — in fact, even unrelated typing contexts. This flexibility is useful to when relating closures of type $\sigma \to \tau$: two closures might be related even if they have close over environments of different shape. For instance, closures $v_1 = \varnothing\ [\lambda x \to 0]$ and $v_2 = (y := 0)\ [\lambda x \to y]$ are related by a nil change such as $dv = \varnothing\ [\lambda x\ dx \to 0]$. In Appendix C.5, we discuss instead an *intensional* definition of validity.

In particular, for function types the relation $\mathcal{RV}[\![\, \sigma \to \tau \,]\!]$ relates function values f_1, df and f_2 if they map *related input values* (and for df input changes) to *related output computations*.

We also extend the relation on values to environments via $\mathcal{RG}[\![\, \Gamma \,]\!]$: environments are related if their corresponding entries are related values.

$$
\begin{aligned}
\mathcal{RV}[\![\, \mathbb{N} \,]\!] &= \{\, (n_1, dn, n_2) \mid n_1, dn, n_2 \in \mathbb{N} \text{ and } n_1 + dn = n_2 \,\} \\
\mathcal{RV}[\![\, \tau_a \times \tau_b \,]\!] &= \{\, (\langle v_{a1}, v_{b1} \rangle, \langle dv_a, dv_b \rangle, \langle v_{a2}, v_{b2} \rangle) \mid \\
&\qquad (v_{a1}, dv_a, v_{a2}) \in \mathcal{RV}[\![\, \tau_a \,]\!] \text{ and } (v_{b1}, dv_b, v_{b2}) \in \mathcal{RV}[\![\, \tau_b \,]\!] \,\} \\
\mathcal{RV}[\![\, \sigma \to \tau \,]\!] &= \{\, (v_{f1}, dv_f, v_{f2}) \mid \\
&\qquad \vDash v_{f1} : \sigma \to \tau \text{ and } \vDash_\Delta dv_f : \sigma \to \tau \text{ and } \vDash v_{f2} : \sigma \to \tau \\
&\qquad \text{and} \\
&\qquad \forall (v_1, dv, v_2) \in \mathcal{RV}[\![\, \sigma \,]\!]\,. \\
&\qquad\quad (\mathrm{app}\ v_{f1}\ v_1, \mathrm{dapp}\ dv_f\ v_1\ dv, \mathrm{app}\ v_{f2}\ v_2) \in \mathcal{RC}[\![\, \tau \,]\!] \,\} \\
\mathcal{RC}[\![\, \tau \,]\!] &= \{\, (\rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \mid \\
&\qquad (\exists \Gamma_1\ \Gamma\ \Gamma_2.\ \Gamma_1 \vdash t_1 : \tau \text{ and } \Gamma \vdash_\Delta dt : \tau \text{ and } \Gamma_2 \vdash t_2 : \tau) \\
&\qquad \text{and} \\
&\qquad \forall v_1\ v_2. \\
&\qquad\quad (\rho_1 \vdash t_1 \Downarrow v_1 \text{ and } \rho_2 \vdash t_2 \Downarrow v_2) \Rightarrow \\
&\qquad\quad \exists dv.\ \rho \bullet d\rho \vdash_\Delta dt \Downarrow dv \text{ and } (v_1, dv, v_2) \in \mathcal{RV}[\![\, \tau \,]\!] \,\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{RG}[\![\, \varepsilon \,]\!] &= \{\, (\varnothing, \varnothing, \varnothing) \,\} \\
\mathcal{RG}[\![\, \Gamma, x : \tau \,]\!] &= \{\, ((\rho_1, x := v_1), (d\rho, dx := dv), (\rho_2, x := v_2)) \mid \\
&\qquad (\rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\, \Gamma \,]\!] \text{ and } (v_1, dv, v_2) \in \mathcal{RV}[\![\, \tau \,]\!] \,\} \\
\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 : \tau &= \forall (\rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\, \Gamma \,]\!]\,. \\
&\qquad (\rho_1 \vdash t_1, \rho_1 \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\, \tau \,]\!]
\end{aligned}
$$

Figure C.4: Defining extensional validity via logical relations and big-step semantics.

Given these definitions, one can prove the fundamental property.

Theorem C.3.1 (Fundamental property: correctness of $\mathcal{D}[\![-]\!]$)
For every well-typed term $\Gamma \vdash t : \tau$ we have that $\Gamma \models \mathcal{D}[\![t]\!] \blacktriangleright t \hookrightarrow t : \tau$. □

Proof sketch. By induction on the structure on terms, using ideas similar to Theorem 12.2.2. □

It also follows that $\mathcal{D}[\![t]\!]$ normalizes:

Corollary C.3.2 ($\mathcal{D}[\![-]\!]$ normalizes to a nil change)
For any well-typed and closed term $\vdash t : \tau$, there exist value v and change value dv such that $\vdash t \Downarrow v$, $\vdash_\Delta \mathcal{D}[\![t]\!] \Downarrow dv$ and $(v, dv, v) \in \mathcal{RV}[\![\tau]\!]$. □

Proof. A corollary of the fundamental property and of the normalization theorem (Theorem C.1.3): since t is well-typed and closed it normalizes to a value v for some step count ($\vdash t \Downarrow v$). The empty environment change is valid: $(\varnothing, \varnothing, \varnothing) \in \mathcal{RG}[\![\varepsilon]\!]$, so from the fundamental property we get that $(\vdash t, \vdash_\Delta \mathcal{D}[\![t]\!], \vdash t) \in \mathcal{RC}[\![\tau]\!]$. From the definition of $\mathcal{RC}[\![\tau]\!]$ and $\vdash t \Downarrow v$ it follows that there exists dv such that $\vdash_\Delta \mathcal{D}[\![t]\!] \Downarrow dv$ and $(v, dv, v) \in \mathcal{RV}[\![\tau]\!]$. □

Remark C.3.3
Compared to prior work, these relations are unusual for two reasons. First, instead of just relating two executions of a term, we relate two executions of a term with an execution of a change term. Second, most such logical relations (including Ahmed [2006]'s one, but except Acar et al. [2008]'s one) define a logical relation (sometimes called *logical equivalence*) that characterizes contextual equivalence, while we don't.

Consider a logical equivalence defined through sets $\mathcal{RC}[\![\tau]\!]$ and $\mathcal{RV}[\![\tau]\!]$. If $(t_1, t_2) \in \mathcal{RC}[\![\tau]\!]$ holds and t_1 terminates (with result v_1), then t_2 must terminate as well (with result v_2), and their results v_1 and v_2 must in turn be logically equivalent ($v_1, v_2 \in \mathcal{RV}[\![\tau]\!]$). And at base types like $\mathbb{N}$, $(v_1, v_2) \in \mathcal{RV}[\![\mathbb{N}]\!]$ means that $v_1 = v_2$.

Here. instead, the fundamental property relates two executions of a term on *different* inputs, which might take different paths during execution. In a suitably extended language, we could even write term $t = \lambda x \to$ **if** $x = 0$ **then** 1 **else** *loop* and run it on inputs $v_1 = 0$ and $v_2 = 1$: these inputs are related by change $dv = 1$, but t will converge on v_1 and diverge on v_2. We must use a semantics that allow such behavioral difference. Hence, at base type $\mathbb{N}$, $(v_1, dv, v_2) \in \mathcal{RV}[\![\mathbb{N}]\!]$ means just that dv is a change from v_1 to v_2, hence that $v_1 \oplus dv$ is equivalent to v_2 because $\oplus$ agrees with extensional validity in this context as well. And if $(\rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\tau]\!]$, term t_1 might converge while t_2 diverges: only if both converge must their results be related.

These subtleties become more relevant in the presence of general recursion and non-terminating programs, as in untyped language λ_A, or in a hypothetical extension of $\lambda_{A\to}$ with fixpoint operators. □

C.4 Step-indexed extensional validity ($\lambda_{A\to}$, $\lambda^\Delta_{A\to}$)

Step-indexed logical relations define approximations to a relation, to enable dealing with non-terminating programs. Logical relations relate the behavior of multiple terms during evaluation; with step-indexed logical relations, we can take a bound k and restrict attention to evaluations that take at most k steps overall, as made precise by the definitions. Crucially, entities related at step count k are also related at any step count $j < k$. Conversely, the higher the step count, the more precise the defined relation. In the limit, if entities are related at all step counts, we simply say they are related. This construction of limits resembles various other approaches to constructing relations by approximations, but the entire framework remains elementary. In particular, the relations are defined simply because they are well-founded (that is, only defined by induction on smaller numbers).

Proofs of logical relation statement need to deal with step-indexes, but when they work (as here) they are otherwise not much harder than other syntactic or logical relation proofs.

For instance, if we define equivalence as a step-indexed logical relation, we can say that two terms are equivalent for k or fewer steps, even if they might have different behavior with more steps available. In our case, we can say that a change appears valid at step count k if it behaves like a valid change in "observations" using at most k steps.

Like before, we define a relation on values and one on computations as sets $\mathcal{RV}[\![\ \tau\ ]\!]$ and $\mathcal{RC}[\![\ \tau\ ]\!]$. Instead of indexing the sets with the step-count, we add the step counts to the tuples they contain: so for instance $(k, v_1, dv, v_2) \in \mathcal{RV}[\![\ \tau\ ]\!]$ means that value v_1, change value dv and value v_2 are related at step count k (or k-related), and similarly for $\mathcal{RC}[\![\ \tau\ ]\!]$.

The details or the relation definitions are subtle, but follow closely the use of step-indexing by Acar, Ahmed, and Blume [2008]. We add mention of changes, and must decide how to use step-indexing for them.

How step-indexing proceeds We explain gradually in words how the definition proceeds.

First, we say k-related function values take j-related arguments to j-related results for all j less than k. That is reflected in the definition for $\mathcal{RV}[\![\ \sigma \to \tau\ ]\!]$: it contains $(k, v_{f1}, dv_f, v_{f2})$ if, for all $(j, v_1, dv, v_2) \in \mathcal{RV}[\![\ \sigma\ ]\!]$) with $j < k$, the result of applications are also j-related. However, the result of application are not syntactic applications encoding $v_{f1}\ v_1$[4]. It is instead necessary to use app $v_{f1}\ v_1$, the result of one step of reduction. The two definitions are not equivalent because a syntactic application would take one extra step to reduce.

The definition for computations takes longer to describe. Roughly speaking, computations are k-related if, after j steps of evaluations (with $j < k$), they produce values related at $k - j$ steps; in particular, if the computations happen to be neutral forms and evaluate in zero steps, they're k-related as computations if the values they produce are k-related. In fact, the rule for evaluation has a wrinkle. Formally, instead of saying that computations $(\rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2)$ are k-related, we say that $(k, \rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\ \tau\ ]\!]$. We do not require all three computations to take j steps. Instead, if the first computation $\rho_1 \vdash t_1$ evaluates to a value in $j < k$ steps, and the second computation $\rho_2 \vdash t_2$ evaluates to a value in any number of steps, *then* the change computation $\rho \bullet d\rho \vdash_\Delta dt$ must also terminate to a change value dv (in an unspecified number of steps), and the resulting values must be related at step-count $k - j$ (that is, $(k - j, v_1, dv, v_2) \in \mathcal{RV}[\![\ \tau\ ]\!]$).

What is new in the above definition is the addition of changes, and the choice to allow change term dt to evaluate to dv in an unbounded number of steps (like t_2), as no bound is necessary for our proofs. This is why the semantics we defined for change terms has no step counts.

Well-foundedness of step-indexed logical relations has a small wrinkle, because $k - j$ need not be strictly less than k. But we define the two relations in a mutually recursive way, and the pair of relations at step-count k is defined in terms of the pair of relation at smaller step-count. All other recursive uses of relations are at smaller step-indexes.

In this section we index the relation by both types and step-indexes, since this is the one we use in our mechanized proof. This relation is defined by structural induction on types. We show this definition in Fig. C.5. Instead, in Appendix C.6 we consider untyped λ-calculus and drop types. The resulting definition is very similar, but is defined by well-founded recursion on step-indexes.

Again, since we use Church typing and only mechanize typed terms, we must include in all cases appropriate typing assumptions. This choice does not match Ahmed [2006] but is one alternative she describes as equivalent. Indeed, while adapting the proof the extra typing assumptions and proof obligations were not a problem.

[4]That happens to be illegal syntax in this presentation, but can be encoded for instance as $f := v_{f1}, x := v_1 \vdash (f\ x)$; and the problem is more general.

$$
\begin{aligned}
\mathcal{RV}[\![\,\mathbb{N}\,]\!] &= \{\,(k, n_1, dn, n_2) \mid n_1, dn, n_2 \in \mathbb{N} \text{ and } n_1 + dn = n_2\,\} \\
\mathcal{RV}[\![\,\tau_a \times \tau_b\,]\!] &= \{\,(k, \langle v_{a1}, v_{b1}\rangle, \langle dv_a, dv_b\rangle, \langle v_{a2}, v_{b2}\rangle) \mid \\
&\qquad (k, v_{a1}, dv_a, v_{a2}) \in \mathcal{RV}[\![\,\tau_a\,]\!] \text{ and } (k, v_{b1}, dv_b, v_{b2}) \in \mathcal{RV}[\![\,\tau_b\,]\!]\,\} \\
\mathcal{RV}[\![\,\sigma \to \tau\,]\!] &= \{\,(k, v_{f1}, dv_f, v_{f2}) \mid \\
&\qquad \vDash v_{f1} : \sigma \to \tau \text{ and } \vDash_\Delta dv_f : \sigma \to \tau \text{ and } \vDash v_{f2} : \sigma \to \tau \\
&\qquad \text{and} \\
&\qquad \forall (j, v_1, dv, v_2) \in \mathcal{RV}[\![\,\sigma\,]\!]\,.\ j < k \Rightarrow \\
&\qquad\quad (j, \mathsf{app}\ v_{f1}\ v_1, \mathsf{dapp}\ dv_f\ v_1\ dv, \mathsf{app}\ v_{f2}\ v_2) \in \mathcal{RC}[\![\,\tau\,]\!]\,\} \\
\mathcal{RC}[\![\,\tau\,]\!] &= \{\,(k, \rho_1 \vdash t_1, \rho \blacklozenge d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \mid \\
&\qquad (\exists \Gamma_1\ \Gamma\ \Gamma_2.\ \Gamma_1 \vdash t_1 : \tau \text{ and } \Gamma \vdash_\Delta dt : \tau \text{ and } \Gamma_2 \vdash t_2 : \tau) \\
&\qquad \text{and} \\
&\qquad \forall j\ v_1\ v_2. \\
&\qquad\quad (j < k \text{ and } \rho_1 \vdash t_1 \Downarrow_j v_1 \text{ and } \rho_2 \vdash t_2 \Downarrow v_2) \Rightarrow \\
&\qquad\quad \exists dv.\ \rho \blacklozenge d\rho \vdash_\Delta dt \Downarrow dv \text{ and } (k - j, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]\,\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{RG}[\![\,\varepsilon\,]\!] &= \{\,(k, \varnothing, \varnothing, \varnothing)\,\} \\
\mathcal{RG}[\![\,\Gamma, x : \tau\,]\!] &= \{\,(k, (\rho_1, x := v_1), (d\rho, dx := dv), (\rho_2, x := v_2)) \mid \\
&\qquad (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\,\Gamma\,]\!] \text{ and } (k, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]\,\} \\
\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 : \tau &= \forall (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\,\Gamma\,]\!]\,. \\
&\qquad (k, \rho_1 \vdash t_1, \rho_1 \blacklozenge d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\,\tau\,]\!]
\end{aligned}
$$

Figure C.5: Defining extensional validity via *step-indexed* logical relations and big-step semantics.

At this moment, we do not require that related closures contain related environments: again, we are defining *extensional* validity.

Given these definitions, we can prove that all relations are *downward-closed*: that is, relations at step-count n imply relations at step-count $k < n$.

Lemma C.4.1 (Extensional validity is downward-closed)
Assume $k \le n$.

1. If $(n, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]$ then $(k, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]$.
2. If $(n, \rho_1 \vdash t_1, \rho \blacklozenge d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\,\tau\,]\!]$ then
$$(k, \rho_1 \vdash t_1, \rho \blacklozenge d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}[\![\,\tau\,]\!]\,.$$
3. If $(n, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\,\Gamma\,]\!]$ then $(k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\,\Gamma\,]\!]$. □

Proof sketch. For $\mathcal{RV}[\![\,\tau\,]\!]$, case split on τ and expand hypothesis and thesis. If $\tau = \mathbb{N}$ they coincide. For $\mathcal{RV}[\![\,\sigma \to \tau\,]\!]$, parts of the hypothesis and thesis match. For some relation P, the rest of the hypothesis has shape $\forall j < n.\ P\ (j, v_1, dv, v_2)$ and the rest of the thesis has shape $\forall j < k.\ P\ (j, v_1, dv, v_2)$. Assume $j < k$. We must prove $P\ (j, v_1, dv, v_2)$, but since $j < k \le n$ we can just apply the hypothesis.

The proof for $\mathcal{RC}[\![\,\tau\,]\!]$ follows the same idea as $\mathcal{RV}[\![\,\sigma \to \tau\,]\!]$.
For $\mathcal{RG}[\![\,\Gamma\,]\!]$, apply the theorem for $\mathcal{RV}[\![\,\tau\,]\!]$ to each environments entry $x : \tau$. □

At this point, we prove the fundamental property.

Theorem C.4.2 (Fundamental property: correctness of $\mathcal{D}[\![-]\!]$)
For every well-typed term $\Gamma \vdash t : \tau$ we have that $\Gamma \models \mathcal{D}[\![\,t\,]\!] \blacktriangleright t \hookrightarrow t : \tau$. □

Proof sketch. By structural induction on typing derivations, using ideas similar to Theorem C.3.1 and relying on Lemma C.4.1 to reduce step counts where needed. □

C.5 Step-indexed intensional validity

Up to now, we have defined when a function change is valid *extensionally*, that is, purely based on its behavior, as we have done earlier when using denotational semantics. We conjecture that with these one can define $\oplus$ and prove it agrees with extensional validity. However, we have not done so.

Instead, we modify definitions in Appendix C.4 to define validity *intensionally*. To ensure that $f_1 \oplus df = f_2$ (for a suitable $\oplus$) we choose to require that closures f_1, df and f_2 close over environments of matching shapes. This change does not complicate the proof of the fundamental lemma: all the additional proof obligations are automatically satisfied.

However, it can still be necessary to replace a function value with a different one. Hence we extend our definition of values to allow replacement values. Closure replacements produce replacements as results, so we make replacement values into valid changes for all types. We must also extend the change semantics, both to allow evaluating closure replacements, and to allow derivatives of primitive to handle replacement values.

We have not added replacement values to the syntax, so currently they can just be added to change environments, but we don't expect adding them to the syntax would cause any significant trouble.

We present the changes described above for the typed semantics. We have successfully mechanized this variant of the semantics as well. Adding replacement values $!v$ requires extending the definition of change values, evaluation and validity. We add replacement values to change values:

$$dv := \ldots \mid\ !v$$

Derivatives of primitives, when applied to replacement changes, must recompute their output. The required additional equations are not interesting, but we show them anyway for completeness:

$$\begin{aligned}
&\mathcal{P}_\Delta[\![\,\mathbf{succ}\,]\!]\ n_1\ (!n_2) &&= !(n_2 + 1)\\
&\mathcal{P}_\Delta[\![\,\mathbf{add}\,]\!]\ \langle _, _ \rangle\ (!\langle m_2, n_2 \rangle) &&= !(m_2 + n_2)\\
&\mathcal{P}_\Delta[\![\,\mathbf{add}\,]\!]\ p_1\ (dp\ @\ \langle dm, !n_2 \rangle) &&= !(\mathcal{P}[\![\,\mathbf{add}\,]\!]\ (p_1 \oplus dp))\\
&\mathcal{P}_\Delta[\![\,\mathbf{add}\,]\!]\ p_1\ (dp\ @\ \langle !m, dv \rangle) &&= !(\mathcal{P}[\![\,\mathbf{add}\,]\!]\ (p_1 \oplus dp))
\end{aligned}$$

Evaluation requires a new rule, E-BangApp, to evaluate change applications where the function change evaluates to a replacement change:

$$\frac{\rho \blacklozenge d\rho \vdash_\Delta dw_f \Downarrow\ !v_f \qquad \rho \vdash w_a \Downarrow v_a \qquad \rho \blacklozenge d\rho \vdash_\Delta dw_a \Downarrow dv_a \qquad \mathsf{app}\ v_f\ (v_a \oplus dv_a) \Downarrow v}{\rho \blacklozenge d\rho \vdash_\Delta dw_f\ w_a\ dw_a \Downarrow\ !v}\ \text{E-BangApp}$$

Evaluation rule E-BangApp requires defining $\oplus$ on syntactic values. We define it *intensionally*:

Definition C.5.1 (Update operator ⊕)
Operator ⊕ is defined on values by the following equations, where match` ρ $d\rho$ (whose definition we omit) tests if ρ and $d\rho$ are environments for the same typing context Γ.

$v_1 \oplus !v_2 = v_2$
$\rho\ [\lambda x \to t] \oplus d\rho\ [\lambda x\ dx \to dt] =$
  if match` ρ $d\rho$ **then**
    -- If ρ and $d\rho$ are environments for the same typing context Γ:
    $(\rho \oplus d\rho)\ [\lambda x \to t]$
  else
    -- otherwise, the input change is invalid, so just give
    -- any type-correct result:
    $\rho\ [\lambda x \to t]$
$n \oplus dn = n + dn$
$\langle v_{a1}, v_{b1}\rangle \oplus \langle dv_a, dv_b\rangle = \langle v_{a1} \oplus dv_a, v_{b1} \oplus dv_b\rangle$
  -- An additional equation is needed in the untyped language,
  -- not in the typed one. This equation is for invalid
  -- changes, so we can just return v_1:
$v_1 \oplus dv = v_1$

We define ⊕ on environments for matching contexts to combine values and changes pointwise:

$(x_1 := v_1, \ldots, x_n := v_n) \oplus (dx_1 := dv_1, \ldots, dx_n := dv_n) =$
  $(x_1 := v_1 \oplus dv_1, \ldots, x_n := v_n \oplus dv_n)$

The definition of update for closures can only update them in few cases, but this is not a problem: as we show in a moment, we restrict validity to the closure changes for which it is correct.

We ensure replacement values are accepted as valid for all types, by requiring the following equation holds (hence, modifying all equations for $\mathcal{RV}[\![-]\!]$; we omit details):

$$\mathcal{RV}[\![\,\tau\,]\!] \supseteq \{\,(k, v_1, !v_2, v_2) \mid \vDash v_1 : \tau \ \text{ and } \ \vDash v_2 : \tau\,\} \tag{C.1}$$

where we write $\vDash v : \tau$ to state that value v has type τ; we omit the unsurprising rules for this judgement.

To restrict valid closure changes, $\mathcal{RV}[\![\,\sigma \to \tau\,]\!]$ now requires that related elements satisfy predicate *matchImpl* $v_{f1}\ dv_f\ v_{f2}$, defined by the following equations:

matchImpl
  $(\rho_1\ [\lambda x \to t])$
  $(\rho_1 \blacklozenge d\rho\ [\lambda x\ dx \to \mathcal{D}[\![\,t\,]\!]])$
  $((\rho_1 \oplus d\rho)\ [\lambda x \to t]) = \mathit{True}$
matchImpl _ _ _ = *False*

In other words, validity $(k, v_{f1}, dv_f, v_{f2}) \in \mathcal{RV}[\![\,\sigma \to \tau\,]\!]$ now requires via matchImpl that the base closure environment ρ_1 and the base environment of the closure change dv_f coincide, that $\rho_2 = \rho_1 \oplus d\rho$, and that v_{f1} and v_{f2} have $\lambda x \to t$ as body while dv_f has body $\lambda x\ dx \to \mathcal{D}[\![\,t\,]\!]$.

To define intensional validity for function changes, $\mathcal{RV}[\![\,\sigma \to \tau\,]\!]$ must use matchImpl and

explicitly support replacement closures to satisfy Eq. (C.1). Its definition is as follows:

$$\begin{aligned}\mathcal{RV}[\![\,\sigma \to \tau\,]\!] = \{\,(k, v_{f1}, dv_f, v_{f2}) \mid\ & \\ & \vDash v_{f1} : \sigma \to \tau \text{ and } \vDash_\Delta dv_f : \sigma \to \tau \text{ and } \vDash v_{f2} : \sigma \to \tau \\ & \text{and} \\ & \mathsf{matchImpl}\ v_{f1}\ dv_f\ v_{f2} \\ & \text{and} \\ & \forall (j, v_1, dv, v_2) \in \mathcal{RV}[\![\,\sigma\,]\!]\ .\ j < k \Rightarrow \\ & \quad (j, \mathsf{app}\ v_{f1}\ v_1, \mathsf{dapp}\ dv_f\ v_1\ dv, \mathsf{app}\ v_{f2}\ v_2) \in \mathcal{RC}[\![\,\tau\,]\!]\,\} \cup \\ & \{\,(k, v_{f1}, !v_{f2}, v_{f2}) \mid \vDash v_{f1} : \sigma \to \tau \text{ and } \vDash v_{f2} : \sigma \to \tau\,\}\end{aligned}$$

Definitions of $\mathcal{RV}[\![-]\!]$ for other types can be similarly updated to support replacement changes and satisfy Eq. (C.1).

Using these updated definitions, we can again prove the fundamental property, with the same statement as Theorem C.4.2. Furthermore, we now prove that $\oplus$ agrees with validity.

Theorem C.5.2 (Fundamental property: correctness of $\mathcal{D}[\![-]\!]$)
For every well-typed term $\Gamma \vdash t : \tau$ we have that $\Gamma \models \mathcal{D}[\![\,t\,]\!] \blacktriangleright t \hookrightarrow t : \tau$. □

Theorem C.5.3 ($\oplus$ agrees with step-indexed intensional validity)
If $\forall k.\, (k, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]$ then $v_1 \oplus dv = v_2$. □

Proof. In this system the thesis holds, in fact, even if we only assume $(k, v_1, dv, v_2) \in \mathcal{RV}[\![\,\tau\,]\!]$. So we pick an arbitrary k.

The proof then proceeds by induction on types. For type $\mathbb{N}$, validity coincides with the thesis. For type $\langle \tau_a, \tau_b \rangle$, we must apply the induction hypothesis on both pair components.

For closures, validity requires that $v_1 = \rho_1\ [\lambda x \to t], dv = d\rho\ [\lambda x\ dx \to \mathcal{D}[\![\,t\,]\!]], v_2 = \rho_2\ [\lambda x \to t]$ with $\rho_1 \oplus d\rho = \rho_2$, and there exists Γ such that $\Gamma, x : \sigma \vdash t : \tau$. Moreover, from validity we can show that ρ and $d\rho$ have matching shapes: ρ is an environment matching Γ and $d\rho$ is a change environment matching $\Delta\Gamma$. Hence, $v_1 \oplus dv$ can update the stored environment, and we can show the thesis by the following calculation:

$$v_1 \oplus dv = \rho_1\ [\lambda x \to t] \oplus d\rho\ [\lambda x\ dx \to dt] = \\ (\rho_1 \oplus d\rho)\ [\lambda x \to t] = \rho_2\ [\lambda x \to t] = v_2 \qquad \square$$

As mentioned, Theorem C.5.3 would hold even if it only required validity (k, v_1, dv, v_2) to hold at a particular step-index. But we still state a weaker version requiring validity at all step indexes; we conjecture that for other systems we consider in this chapter, requiring validity at all step-indexes is necessary. For instance, step-indexed extensional validity for function types at index 0 is vacuously true and so can't agree with $\oplus$, because it is only defined in terms of validity at step-indexes smaller than 0 (which do not exist).

Nil changes We can define **0** intensionally, as a metafunction on values and environments, and prove it correct. For closures, we differentiate the body and recurse on the environment. The definition extends from values to environments variable-wise, so we omit the standard formal definition.

Definition C.5.4 (Nil changes 0)
Nil changes on values are defined as follows:

$$\mathbf{0}_{\rho\,[\lambda x \to t]} = \rho \blacklozenge \mathbf{0}_\rho\ [\lambda x\ dx \to \mathcal{D}[\![\, t \,]\!]]$$
$$\mathbf{0}_{\langle a,b\rangle} = \langle \mathbf{0}_a, \mathbf{0}_b \rangle$$
$$\mathbf{0}_n = 0$$

Lemma C.5.5 (0 produces valid changes)
For all values $\vDash v : \tau$ and indexes k, we have $(k, v, \mathbf{0}_v, v) \in \mathcal{RV}[\![\, \tau \,]\!]$. □

Proof sketch. By induction on v. For closures we must apply the fundamental property (Theorem C.5.2) to $\mathcal{D}[\![\, t \,]\!]$. □

Because **0** transforms closure bodies, we cannot define it internally to the language. This problem can be avoided by defunctionalizing functions and function changes, as we do in Appendix D.

We conclude with the overall correctness theorem, analogous to Corollary 13.4.5.

Corollary C.5.6 ($\mathcal{D}[\![-]\!]$ is correct, corollary)
Take any term t that is well-typed ($\Gamma \vdash t : \tau$) and any suitable environments ρ_1, $d\rho$, ρ_2, intensionally valid at any step count ($\forall k.\, (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\, \Gamma \,]\!]$). Assume t terminates in both the old environment ρ_1 and the new environment ρ_2, evaluating to output values v_1 and v_2 ($\rho_1 \vdash t \Downarrow v_1$ and $\rho_2 \vdash t \Downarrow v_2$). Then $\mathcal{D}[\![\, t \,]\!]$ evaluates in environment ρ and change environment $d\rho$ to a change value dv ($\rho_1 \blacklozenge d\rho \vdash_\Delta t \Downarrow dv$), and dv is a valid change from v_1 to v_2, so that $v_1 \oplus dv = v_2$. □

Proof. Follows immediately from Theorem C.5.2 and Theorem C.5.3. □

C.6 Untyped step-indexed validity (λ_A, λ_A^Δ)

By removing mentions of types from step-indexed validity (intensional or extensional, though we show extensional definitions), we can adapt it to an untyped language. We can still distinguish between functions, numbers and pairs by matching on values themselves, instead of matching on types. Without types, typing contexts Γ now degenerate to lists of free variables of a term; we still use them to ensure that environments contain enough valid entries to evaluate a term. Validity applies to terminating executions, hence we need not consider executions producing dynamic type errors when proving the fundamental property.

We show resulting definitions for extensional validity in Fig. C.6; but we can also define intensional validity and prove the fundamental lemma for it. As mentioned earlier, for λ_A we must turn $\mathcal{P}[\![-]\!] -$ and $\mathcal{P}_\Delta[\![-]\!] -$ into relations and update E-PRIM accordingly.

The main difference in the proof is that this time, the recursion used in the relations can only be proved to be well-founded because of the use of step-indexes; we omit details [Ahmed, 2006].

Otherwise, the proof proceeds just as earlier in Appendix C.4: We prove that the relations are downward-closed, just like in Lemma C.4.1 (we omit the new statement), and we prove the new fundamental lemma by induction on the structure of terms (not of typing derivations).

Theorem C.6.1 (Fundamental property: correctness of $\mathcal{D}[\![-]\!]$)
If $FV\ (t) \subseteq \Gamma$ then we have that $\Gamma \models \mathcal{D}[\![\, t \,]\!] \blacktriangleright t \hookrightarrow t$. □

Proof sketch. Similar to the proof of Theorem C.4.2, but by structural induction on terms and complete induction on step counts, not on typing derivations.

However, we can use the induction hypothesis in the same ways as in earlier proofs for typed languages: all uses of the induction hypothesis in the proof are on smaller terms, and some also at smaller step counts. □

$$
\begin{aligned}
\mathcal{RV} = \{\ & (k, n_1, dn, n_2) \mid n_1, dn, n_2 \in \mathbb{N} \text{ and } n_1 + dn = n_2 \ \} \cup \\
& \{\ (k, v_{f1}, dv_f, v_{f2}) \mid \\
& \forall (j, v_1, dv, v_2) \in \mathcal{RV}.\ j < k \Rightarrow \\
& \quad (j, \text{app}\ v_{f1}\ v_1, \text{dapp}\ dv_f\ v_1\ dv, \text{app}\ v_{f2}\ v_2) \in \mathcal{RC}\ \} \cup \\
& \{\ (k, \langle v_{a1}, v_{b1}\rangle, \langle dv_a, dv_b\rangle, \langle v_{a2}, v_{b2}\rangle) \mid \\
& (k, v_{a1}, dv_a, v_{a2}) \in \mathcal{RV} \text{ and } (k, v_{b1}, dv_b, v_{b2}) \in \mathcal{RV}\ \} \\
\mathcal{RC} = \{\ & (k, \rho_1 \vdash t_1, \rho \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \mid \\
& \forall j\ v_1\ v_2. \\
& \quad (j < k \text{ and } \rho_1 \vdash t_1 \Downarrow_j v_1 \text{ and } \rho_2 \vdash t_2 \Downarrow v_2) \Rightarrow \\
& \quad \exists dv.\ \rho \bullet d\rho \vdash_\Delta dt \Downarrow dv \text{ and } (k - j, v_1, dv, v_2) \in \mathcal{RV}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{RG}[\![\ \varepsilon\ ]\!] &= \{\ (k, \varnothing, \varnothing, \varnothing)\ \} \\
\mathcal{RG}[\![\ \Gamma, x\ ]\!] &= \{\ (k, (\rho_1, x := v_1), (d\rho, dx := dv), (\rho_2, x := v_2)) \mid \\
&\qquad (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\ \Gamma\ ]\!] \text{ and } (k, v_1, dv, v_2) \in \mathcal{RV}\ \} \\
\Gamma \models dt \blacktriangleright t_1 \hookrightarrow t_2 &= \forall (k, \rho_1, d\rho, \rho_2) \in \mathcal{RG}[\![\ \Gamma\ ]\!]\ . \\
&\qquad (k, \rho_1 \vdash t_1, \rho_1 \bullet d\rho \vdash_\Delta dt, \rho_2 \vdash t_2) \in \mathcal{RC}
\end{aligned}
$$

Figure C.6: Defining extensional validity via *untyped step-indexed* logical relations and big-step semantics.

C.7 General recursion in $\lambda_{A\to}$ and $\lambda^{\Delta}_{A\to}$

We have sketched informally in Sec. 15.1 how to support fixpoint combinators.

We have also extended our typed languages with a fixpoint combinators and proved them correct formally (not mechanically, yet). In this section, we include the needed definitions to make our claim precise. They are mostly unsurprising, if long to state.

Since we are in a call-by-value setting, we only add recursive functions, not recursive values in general. To this end, we replace λ-abstraction $\lambda x \to t$ with recursive function $\mathbf{rec}\ f\ x \to t$, which binds both f and x in t, and replaces f with the function itself upon evaluation.

The associated small-step reduction rule would be $(\mathbf{rec}\ f\ x \to t)\ v \to t\ [x := v, f := \mathbf{rec}\ f\ x \to t]$. As before, we formalize reduction using big-step semantics.

Typing rule T-Lam is replaced by a rule for recursive functions:

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathbf{rec}\ f\ x \to t : \sigma \to \tau}\ \text{T-Rec}$$

We replace closures with recursive closures in the definition of values:

$$
\begin{aligned}
w &::= \mathbf{rec}\ f\ x \to t \mid \dots \\
v &::= \rho\ [\mathbf{rec}\ f\ x \to t] \mid \dots
\end{aligned}
$$

We also modify the semantics for abstraction and application. Rules E-Val and E-App are unchanged: it is sufficient to adapt the definitions of $\mathcal{V}[\![-]\!]$ – and app, so that evaluation of a function value v_f has access to v_f in the environment.

$$\mathcal{V}[\![\ \mathbf{rec}\ f\ x \to t\ ]\!]\ \rho = \rho\ [\mathbf{rec}\ f\ x \to t]$$
$$\begin{aligned}&\mathsf{app}\ (v_f\ @\ (\rho'\ [\mathbf{rec}\ f\ x \to t]))\ v_a =\\ &\quad (\rho', f := v_f, x := v_a) \vdash t\end{aligned}$$

Like in Haskell, we write $x\ @\ p$ in equations to bind an argument as metavariable x and match it against pattern p.

Similarly, we modify the language of changes, the definition of differentiation, and evaluation metafunctions $\mathcal{V}_\Delta[\![-]\!] -$ and dapp. Since the derivative of a recursive function $f = \mathbf{rec}\ f\ x \to t$ can call the base function, we remember the original function body t in the derivative, together with its derivative $\mathcal{D}[\![\ t\ ]\!]$. This should not be surprising: in Sec. 15.1, where recursive functions are defined using *letrec*, a recursive function f is in scope in the body of its derivative df. Here we use a different syntax, but still ensure that f is in scope in the body of derivative df. The definitions are otherwise unsurprising, if long.

$$dw ::= \mathbf{drec}\ f\ df\ x\ dx \to t \bullet dt \mid \ldots$$
$$dv ::= \rho \bullet d\rho\ [\mathbf{drec}\ f\ df\ x\ dx \to t \bullet dt] \mid \ldots$$
$$\mathcal{D}[\![\ \mathbf{rec}\ f\ x \to t\ ]\!] = \mathbf{drec}\ f\ df\ x\ dx \to t \bullet \mathcal{D}[\![\ t\ ]\!]$$
$$\begin{aligned}&\mathcal{V}_\Delta[\![\ \mathbf{drec}\ f\ df\ x\ dx \to dt\ ]\!]\ \rho\ d\rho =\\ &\quad \rho \bullet d\rho\ [\mathbf{drec}\ f\ df\ x\ dx \to t \bullet dt]\end{aligned}$$
$$\begin{aligned}&\mathsf{dapp}\ (dv_f\ @\ (\rho' \bullet d\rho'\ [\mathbf{rec}\ f\ df\ x\ dx \to dt]))\ v_a\ dv_a =\\ &\quad \mathbf{let}\ v_f = \rho'\ [\mathbf{rec}\ f\ x \to t]\\ &\quad \mathbf{in}\ \ (\rho', f := v_f, x := v_a) \bullet (d\rho', df := dv_f, dx := dv_a) \vdash_\Delta dt\end{aligned}$$

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash t : \tau \qquad \Gamma, f : \sigma \to \tau, x : \sigma \vdash_\Delta dt : \tau}{\Gamma \vdash_\Delta \mathbf{drec}\ f\ df\ x\ dx \to t \bullet dt : \sigma \to \tau}\ \text{T-DREC}$$

We can adapt the proof of the fundamental property to the use of recursive functions.

Theorem C.7.1 (Fundamental property: correctness of $\mathcal{D}[\![-]\!]$)
For every well-typed term $\Gamma \vdash t : \tau$ we have that $\Gamma \models \mathcal{D}[\![\ t\ ]\!] \blacktriangleright t \hookrightarrow t : \tau$. □

Proof sketch. Mostly as before, modulo one interesting difference: To prove the fundamental property for recursive functions at step-count k, this time we must use the fundamental property inductively on the same term, but at step-count $j < k$. This happens because to evaluate $dw = \mathcal{D}[\![\ \mathbf{rec}\ f\ x \to t\ ]\!]$ we evaluate $\mathcal{D}[\![\ t\ ]\!]$ with the value dv for dw in the environment: to show this invocation is valid, we must show dw is itself a valid change. But the step-indexed definition to $\mathcal{RV}[\![\ \sigma \to \tau\ ]\!]$ constrains the evaluation of the body only $\forall j < k$.

C.8 Future work

We have shown that $\oplus$ and $\mathbf{0}$ agree with validity, which we consider a key requirement of a core ILC proof. However, change structures support further operators. We leave operator $\ominus$ for future work, though we are not aware of particular difficulties. However, $\odot$ deserves special attention.

C.8.1 Change composition

We have looked into change composition, and it appears that composition of change expressions is not always valid, but we conjecture that composition of change values preserves validity. Showing

that change composition is valid appears related to showing that Ahmed's logical equivalence is a transitive relation, which is a subtle issue. She only proves transitivity in a typed setting and with a stronger relation, and her proof does not carry over directly; indeed, there is no corresponding proof in the untyped setting of Acar, Ahmed, and Blume [2008].

However, the failure of transitivity we have verified is not overly worrisome: the problem is that transitivity is too strong an expectation in general. Assume that $\Gamma \models de_1 \blacktriangleright e_1 \hookrightarrow e_2$ and $\Gamma \models de_2 \blacktriangleright e_2 \hookrightarrow e_3$, and try to show that $\Gamma \models de_1 \circledcirc de_2 \blacktriangleright e_1 \hookrightarrow e_3$: that is, very roughly and ignoring the environments, we can assume that e_1 and e_3 terminate, and have to show that their result satisfy some properties. To use both our hypotheses, we need to know that e_1, e_2 and e_3 all terminate, but we have no such guaranteed for e_2. Indeed, if e_2 always diverges (because it is, say, the diverging term $\omega = (\lambda x \to x\ x)\ (\lambda x \to x\ x)$), then de_1 and de_2 are vacuously valid. If e_1 and e_3 terminate, we can't expect $de_1 \circledcirc de_2$ to be a change between them. To wit, take $e_1 = 0$, $e_2 = \omega$, $e_3 = 10$, and $de_1 = de_2 = 0$. We can verify that for any Γ we have $\Gamma \models de_1 \blacktriangleright e_1 \hookrightarrow e_2$ and $\Gamma \models de_2 \blacktriangleright e_2 \hookrightarrow e_3$, while $\Gamma \models de_1 \circledcirc de_2 \blacktriangleright e_1 \hookrightarrow e_3$ means the absurd $\Gamma \models 0 \circledcirc 0 \blacktriangleright 0 \hookrightarrow 10$.

A possible fix Does transitivity hold if e_2 terminates? That is not sufficient: we still cannot conclude anything by assuming that $(k, e_1, de_1, e_2) \in \mathcal{RC}[\![\ \tau\ ]\!]$ and $(k, e_2, de_2, e_3) \in \mathcal{RC}[\![\ \tau\ ]\!]$. But like in Ahmed [2006], if e_2 and e_3 are related at all step counts, that is, if $(k, e_1, de_1, e_2) \in \mathcal{RC}[\![\ \tau\ ]\!]$ and $(\forall n.\ (n, e_2, de_2, e_3) \in \mathcal{RC}[\![\ \tau\ ]\!])$, and if additionally e_2 terminates, we conjecture that Ahmed's proof goes through. We have however not yet examined all the details.

C.9 Development history

The proof strategy used in this chapter comes from a collaboration between me and Yann Régis-Gianas, who came up with the general strategy and the first partial proofs for untyped λ-calculi. After we both struggled for a while to set up step-indexing correctly enough for a full proof, I first managed to give the definitions in this chapter and complete the proofs here described. Régis-Gianas then mechanized a variant of our proof for untyped λ-calculus in Coq [Giarrusso et al., 2019], that appears here in Sec. 17.3. That proof takes a few different choices, and unfortunately strictly speaking neither proof subsumes the other. (1) We also give a non-step-indexed syntactic proof for simply-typed λ-calculus, together with proofs defining validity extensionally. (2) To support remembering intermediate results by conversion to cache-transfer-style (CTS), Régis-Gianas' proof uses a lambda-lifted A'NF syntax instead of plain ANF. (3) Régis-Gianas' formalization adds to change values a single token **0**, which is a valid nil change for all valid values. Hence, if we know a change is nil, we can erase it. As a downside, evaluating $df\ a\ da$ when df is **0** requires looking up f. In our presentation, instead, if f and g are different function values, they have different nil changes. Such nil changes carry information, so they can be evaluated directly, but they cannot be erased. Techniques in Appendix D enable erasing and reconstructing a nil change df for function value f as long as the value of f is available.

C.10 Conclusion

In this chapter we have shown how to construct novel models for ILC by using (step-indexed) logical relations, and have used this technique to deliver a new syntactic proof of correctness for ILC for simply-typed *lambda*-calculus and to deliver the first proof of correctness of ILC for untyped λ-calculus. Moreover, our proof appears rather close to existing logical-relations proofs, hence we believe it should be possible to translate other results to ILC theorems.

By formally defining intensional validity for closures, we provide a solid foundation for the use of defunctionalized function changes (Appendix D).

This proof builds on Ahmed [2006]'s work on step-indexed logical relations, which enable handling of powerful semantics feature using rather elementary techniques. The only downside is that it can be tricky to set up the correct definitions, especially for a slightly non-standard semantics like ours. As an exercise, we have shown that the our semantics is equivalent to more conventional presentations, down to the produced step counts.

Appendix D

Defunctionalizing function changes

In Chapter 12, and throughout most of Part II, we represent function changes as functions, which can only be applied. However, incremental programs often inspect changes to decide how to react to them most efficiently. Also inspecting function changes would help performance further. Representing function changes as closures, as we do in Chapter 17 and Appendix C, allows implementing some operations more efficient, but is not fully satisfactory. In this chapter, we address these restrictions by *defunctionalizing* functions and function changes, so that we can inspect both at runtime without restrictions.

Once we defunctionalize function changes, we can detect at runtime whether a function change is nil. As we have mentioned in Sec. 16.3, nil function changes can typically be handled more efficiently. For instance, consider $t = map\ f\ xs$, a term that maps function f to each element of sequence xs. In general, $\mathcal{D}[\![\ t\ ]\!] = dmap\ f\ df\ xs\ dxs$ must handle any change in dxs (which we assume to be small) but also apply function change df to each element of xs (which we assume to be big). However, if df is nil we can skip this step, decreasing time complexity of $dmap\ f\ df\ xs\ dxs$ from $O\ (|xs| + |dxs|)$ to $O\ (|dxs|)$.

We will also present a change structure on defunctionalized function changes, and show that operations on defunctionalized function changes become cheaper.

D.1 Setup

We write incremental programs based on ILC by manually writing Haskell code, containing both manually-written plugin code, and code that is transformed systematically, based on informal generalizations and variants of $\mathcal{D}[\![-]\!]$. Our main goal is to study variants of differentiation and of encodings in Haskell, while also studying language plugins for non-trivial primitives, such as different sorts of collections. We make liberal use of GHC extensions where needed.

Code in this chapter has been extracted and type-checked, though we elide a few details (mostly language extensions and imports from the standard library). Code in this appendix is otherwise self-contained. We have also tested a copy of this code more extensively.

As sketched before, we define change structure inside Haskell.

```
class ChangeStruct t where
    -- Next line declares Δt as an injective type function
```

```
  type Δt = r | r → t
  (⊕) :: t → Δt → t
  oreplace :: t → Δt
class ChangeStruct t ⇒ NilChangeStruct t where
  0 :: t → Δt
class ChangeStruct a ⇒ CompChangeStruct a where
    -- Compose change dx₁ with dx₂, so that
    -- x ⊕ (dx₁ ⊚ dx₂) ≡ x ⊕ dx₁ ⊕ dx₂.
  (⊚) :: Δa → Δa → Δa
```

With this code we define type classes *ChangeStruct*, *NilChangeStruct* and *CompChangeStruct*. We explain each of the declared members in turn.

First, type Δt represents the change type for type t. To improve Haskell type inference, we declare that Δ is injective, so that $\Delta t_1 = \Delta t_2$ implies $t_1 = t_2$. This forbids some potentially useful change structures, but in exchange it makes type inference vastly more usable.

Next, we declare $\oplus$, $\mathbf{0}$ and $\circledcirc$ as available to object programs. Last, we introduce *oreplace* to construct replacement changes, characterized by the *absorption law* $x \oplus \mathit{oreplace}\ y = y$ for all x. Function *oreplace* encodes $!t$, that is the bang operator. We use a different notation because ! is reserved for other purposes in Haskell.

These typeclasses omit operation $\ominus$ intentionally: we do *not* require that change structures support a proper difference operation. Nevertheless, as discussed $b \ominus a$ can be expressed through *oreplace b*.

We can then differentiate Haskell functions — even polymorphic ones. We show a few trivial examples to highlight how derivatives are encoded in Haskell, especially polymorphic ones.

```
  -- The standard id function:
id :: a → a
id x = x
  -- and its derivative:
did :: a → Δa → Δa
did x dx = dx
instance (NilChangeStruct a, ChangeStruct b) ⇒
         ChangeStruct (a → b) where
  type Δ(a → b) = a → Δa → Δb
  f ⊕ df = λx → f x ⊕ df x 0ₓ
  oreplace f = λx dx → oreplace (f (x ⊕ dx))
instance (NilChangeStruct a, ChangeStruct b) ⇒
         NilChangeStruct (a → b) where
  0_f = oreplace f
  -- Same for apply:
apply :: (a → b) → a → b
apply f x = f x
dapply :: (a → b) → Δ(a → b) → a → Δa → Δb
dapply f df x dx = df x dx
```

Which polymorphism? As visible here, polymorphism does not cause particular problems. However, we only support ML (or prenex) polymorphism, not first-class polymorphism, for two reasons.

First, with first-class polymorphism, we can encode existential types $\exists X.\ T$, and two values v_1, v_2 of the same existential type $\exists X.\ T$ can hide different types T_1, T_2, Hence, a change between v_1 and v_2 requires handling changes between types. While we discuss such topics in Chapter 18, we avoid them here.

Second, prenex polymorphism is a small extension of simply-typed lambda calculus metatheoretically. We can treat prenex-polymorphic definitions as families of monomorphic (hence simply-typed) definitions; to each definition we can apply all the ILC theory we developed to show differentiation is correct.

D.2 Defunctionalization

Defunctionalization is a whole-program transformation that turns a program relying on first-class functions into a first-order program. The resulting program is expressed in a first-order language (often a subset of the original language); closures are encoded by data values, which embed both the closure environment and a tag to distinguish different function. Defunctionalization also generates a function that interprets encoded closures, which we call *applyFun*.

In a typed language, defunctionalization can be done using generalized algebraic datatypes (GADTs) [Pottier and Gauthier, 2004]. Each first-class function of type $\sigma \to \tau$ is replaced by a value of a new GADT $Fun\ \sigma\ \tau$, that represents defunctionalized function values and has a constructor for each different function. If a first-class function t_1 closes over $x :: \tau_1$, the corresponding constructor C_1 will take $x :: \tau_1$ as an argument. The interpreter for defunctionalized function values has type signature $applyFun :: Fun\ \sigma\ \tau \to \sigma \to \tau$. The resulting programs are expressed in a first-order subset of the original programming language. In defunctionalized programs, all remaining functions are first-order top-level functions.

For instance, consider the program on sequences in Fig. D.1.

$$
\begin{aligned}
&successors :: [\mathbb{Z}] \to [\mathbb{Z}]\\
&successors\ xs = map\ (\lambda x \to x + 1)\ xs\\
&nestedLoop :: [\sigma] \to [\tau] \to [(\sigma, \tau)]\\
&nestedLoop\ xs\ ys = concatMap\ (\lambda x \to map\ (\lambda y \to (x, y))\ ys)\ xs\\
&map :: (\sigma \to \tau) \to [\sigma] \to [\tau]\\
&map\ f\ [\,] = [\,]\\
&map\ f\ (x : xs) = f\ x : map\ f\ xs\\
&concatMap :: (\sigma \to [\tau]) \to [\sigma] \to [\tau]\\
&concatMap\ f\ xs = concat\ (map\ f\ xs)
\end{aligned}
$$

Figure D.1: A small example program for defunctionalization.

In this program, the first-class function values arise from evaluating the three terms $\lambda y \to (x, y)$, that we call *pair*, $\lambda x \to map\ (\lambda y \to (x, y))\ ys$, that we call *mapPair*, and $\lambda x \to x + 1$, that we call *addOne*. Defunctionalization creates a type $Fun\ \sigma\ \tau$ with a constructor for each of the three terms, respectively *Pair*, *MapPair* and *AddOne*. Both *pair* and *mapPair* close over some free variables, so their corresponding constructors will take an argument for each free variable; for *pair* we have

$$x :: \sigma \vdash \lambda y \to (x, y) :: \tau \to (\sigma, \tau),$$

while for *mapPair* we have

$$ys :: [\tau] \vdash \lambda x \to map\ (\lambda y \to (x, y))\ ys :: \sigma \to [(\sigma, \tau)].$$

Hence, the type of defunctionalized functions $Fun\ \sigma\ \tau$ and its interpreter *applyFun* become:

```
data Fun σ τ where
  AddOne  ::          Fun ℤ ℤ
  Pair    :: σ    → Fun τ (σ, τ)
  MapPair :: [τ] → Fun σ [(σ, τ)]

applyFun :: Fun σ τ →    σ → τ
applyFun   AddOne        x = x + 1
applyFun   (Pair x)      y = (x, y)
applyFun   (MapPair ys) x = mapDF (Pair x) ys
```

We need to also transform the rest of the program accordingly.

```
successors :: [ℤ] → [ℤ]
successors xs = map (λx → x + 1) xs

nestedLoopDF :: [σ] → [τ] → [(σ, τ)]
nestedLoopDF xs ys = concatMapDF (MapPair ys) xs

mapDF :: Fun σ τ → [σ] → [τ]
mapDF f [] = []
mapDF f (x : xs) = applyFun f x : mapDF f xs

concatMapDF :: Fun σ [τ] → [σ] → [τ]
concatMapDF f xs = concat (mapDF f xs)
```

Figure D.2: Defunctionalized program.

Some readers might notice this program still uses first-class function, because it encodes multi-argument functions through currying. To get a fully first-order program, we encode multi-arguments functions using tuples instead of currying.[1] Using tuples our example becomes:

```
applyFun :: (Fun σ τ, σ) → τ
applyFun   (AddOne, x)      = x + 1
applyFun   (Pair x, y)      = (x, y)
applyFun   (MapPair ys, x) = mapDF (Pair x, ys)

mapDF :: (Fun σ τ, [σ]) → [τ]
mapDF (f, []) = []
mapDF (f, x : xs) = applyFun (f, x) : mapDF (f, xs)

concatMapDF :: (Fun σ [τ], [σ]) → [τ]
concatMapDF (f, xs) = concat (mapDF (f, xs))

nestedLoopDF :: ([σ], [τ]) → [(σ, τ)]
nestedLoopDF (xs, ys) = concatMapDF (MapPair ys, xs)
```

However, we'll often write such defunctionalized programs using Haskell's typical curried syntax, as in Fig. D.2. Such programs must not contain partial applications.

[1] Strictly speaking, the resulting program is still not first-order, because in Haskell multi-argument data constructors, such as the pair constructor (,) that we use, are still first-class curried functions, unlike for instance in OCaml. To make this program truly first-order, we must formalize tuple constructors as a term constructor, or formalize these function definitions as multi-argument functions. At this point, this discussion is merely a technicality that does not affect our goals, but it becomes relevant if we formalize the resulting first-order language as in Sec. 17.3.

In general, defunctionalization creates a constructor C of type $Fun\ \sigma\ \tau$ for each first-class function expression $\Gamma \vdash t : \sigma \to \tau$ in the source program.[2] While lambda expression l closes *implicitly* over environment $\rho : [\![\Gamma]\!]$, C closes over it explicitly: the values bound to free variables in environment ρ are passed as arguments to constructor C. As a standard optimization, we only includes variables that actually occur free in l, not all those that are bound in the context where l occurs.

D.2.1 Defunctionalization with separate function codes

Next, we show a slight variant of defunctionalization, that we use to achieve our goals with less code duplication, even at the expense of some efficiency; we call this variant *defunctionalization with separate function codes*.

We first encode contexts as types and environments as values. Empty environments are encoded as empty tuples. Environments for a context such as $x :: \tau_1, y :: \tau_2, \ldots$ are encoded as values of type $\tau_1 \times \tau_2 \times \ldots$

In this defunctionalization variant, instead of defining directly a GADT of defunctionalized functions $Fun\ \sigma\ \tau$, we define a GADT of *function codes* $Code\ env\ \sigma\ \tau$, whose values contain no environment. Type *Code* is indexed not just by σ and τ but also by the type of environments, and has a constructor for each first-class function expression in the source program, like $Fun\ \sigma\ \tau$ does in conventional defunctionalization. We then define $Fun\ \sigma\ \tau$ as a pair of a function code of type $Code\ env\ \sigma\ \tau$ and an environment of type env.

As a downside, separating function codes adds a few indirections to the memory representation of closures: for instance we use $(AddOne, ())$ instead of $AddOne$, and $(Pair, 1)$ instead of $Pair\ 1$.

As an upside, with separate function codes we can define many operations generically across all function codes (see Appendix D.2.2), instead of generating definitions matching on each function. What's more, we later define operations that use raw function codes and need no environment; we could alternatively define function codes without using them in the representation of function values, at the expense of even more code duplication. Code duplication is especially relevant because we currently perform defunctionalization by hand, though we are confident it would be conceptually straightforward to automate the process.

Defunctionalizing the program with separate function codes produces the following GADT of function codes:

```
type Env env = (CompChangeStruct env, NilTestable env)
data Code env σ τ where
  AddOne  ::                Code ()    ℤ ℤ
  Pair    ::                Code σ     τ  (σ, τ)
  MapPair :: Env σ ⇒ Code [τ] σ [(σ, τ)]
```

In this definition, type *Env* names a set of typeclass constraints on the type of the environment, using the *ConstraintKinds* GHC language extension. Satisfying these constraints is necessary to implement a few operations on functions. We also require an interpretation function *applyCode* for function codes. If c is the code for a function $f = \lambda x \to t$, calling *applyCode c* computes f's output from an environment *env* for f and an argument for x.

```
applyCode :: Code env σ τ → env → σ → τ
applyCode   AddOne          ()       x =  x + 1
```

[2] We only need codes for functions that are used as first-class arguments, not for other functions, though codes for the latter can be added as well.

$$\begin{array}{llll} \mathit{applyCode} & \mathit{Pair} & x & y = (x, y) \\ \mathit{applyCode} & \mathit{MapPair} & \mathit{ys} & x = \mathit{mapDF}\ (F\ (\mathit{Pair}, x))\ \mathit{ys} \end{array}$$

The implementation of *applyCode MapPair* only works because of the $Env\ \sigma$ constraint for constructor *MapPair*: this constraint is required when constructing the defunctionalized function value that we pass as argument to *mapDF*.

We represent defunctionalized function values through type $RawFun\ env\ \sigma\ \tau$, a type synonym of the product of $Code\ env\ \sigma\ \tau$ and *env*. We encode type $\sigma \to \tau$ through type $Fun\ \sigma\ \tau$, defined as $RawFun\ env\ \sigma\ \tau$ where *env* is existentially bound. We add constraint $Env\ env$ to the definition of $Fun\ \sigma\ \tau$, because implementing $\oplus$ on function changes will require using $\oplus$ on environments.

$$\begin{array}{l} \textbf{type}\ \mathit{RawFun}\ env\ \sigma\ \tau = (\mathit{Code}\ env\ \sigma\ \tau, env) \\ \textbf{data}\ \mathit{Fun}\ \sigma\ \tau\ \textbf{where} \\ \quad F :: \mathit{Env}\ env \Rightarrow \mathit{RawFun}\ env\ \sigma\ \tau \to \mathit{Fun}\ \sigma\ \tau \end{array}$$

To interpret defunctionalized function values, we wrap *applyCode* in a new version of *applyFun*, having the same interface as the earlier *applyFun*.

$$\begin{array}{l} \mathit{applyFun} :: \mathit{Fun}\ \sigma\ \tau \to \sigma \to \tau \\ \mathit{applyFun}\ (F\ (\mathit{code}, env))\ \mathit{arg} = \mathit{applyCode}\ \mathit{code}\ env\ \mathit{arg} \end{array}$$

The rest of the source program is defunctionalized like before, using the new definition of $Fun\ \sigma\ \tau$ and of *applyFun*.

D.2.2 Defunctionalizing function changes

Defunctionalization encodes function values as pairs of function codes and environments. In ILC, a function value can change because the environment changes or because the whole closure is replaced by a different one, with a different function code and different environment. For now, we focus on environment changes for simplicity. To allow inspecting function changes, we defunctionalize them as well, but treat them specially.

Assume we want to defunctionalize a function change *df* with type $\Delta(\sigma \to \tau) = \sigma \to \Delta\sigma \to \Delta\tau$, valid for function $f : \sigma \to \tau$. Instead of transforming type $\Delta(\sigma \to \tau)$ into $Fun\ \sigma\ (Fun\ \Delta\sigma\ \Delta\tau)$, we transform $\Delta(\sigma \to \tau)$ into a new type $DFun\ \sigma\ \tau$, the change type of $Fun\ \sigma\ \tau$ ($\Delta(Fun\ \sigma\ \tau) = DFun\ \sigma\ \tau$). To apply $DFun\ \sigma\ \tau$ we introduce an interpreter $dapplyFun :: Fun\ \sigma\ \tau \to \Delta(Fun\ \sigma\ \tau) \to \Delta(\sigma \to \tau)$, or equivalently $dapplyFun :: Fun\ \sigma\ \tau \to DFun\ \sigma\ \tau \to \sigma \to \Delta\sigma \to \Delta\tau$, which also serves as derivative of *applyFun*.

Like we did for $Fun\ \sigma\ \tau$, we define $DFun\ \sigma\ \tau$ using function codes. That is, $DFun\ \sigma\ \tau$ pairs a function code $Code\ env\ \sigma\ \tau$ together with an environment change and a change structure for the environment type.

$$\textbf{data}\ \mathit{DFun}\ \sigma\ \tau = \forall env.\ \mathit{ChangeStruct}\ env \Rightarrow \mathit{DF}\ (\Delta env, \mathit{Code}\ env\ \sigma\ \tau)$$

Without separate function codes, the definition of *DFun* would have to include one case for each first-class function.

Environment changes

Instead of defining change structures for environments, we encode environments using tuples and define change structures for tuples.

We define first change structures for empty tuples and pairs: Sec. 12.3.3

```
instance ChangeStruct () where
  type Δ() = ()
  _ ⊕ _ = ()
  oreplace _ = ()
instance (ChangeStruct a, ChangeStruct b) ⇒ ChangeStruct (a, b) where
  type Δ(a, b) = (Δa, Δb)
  (a, b) ⊕ (da, db) = (a ⊕ da, b ⊕ db)
  oreplace (a₂, b₂) = (oreplace a₂, oreplace b₂)
```

To define change structures for n-uples of other arities we have two choices, which we show on triples (a, b, c) and can be easily generalized.

We can encode triples as nested pairs $(a, (b, (c, ())))$. Or we can define change structures for triples directly:

```
instance (ChangeStruct a, ChangeStruct b,
          ChangeStruct c) ⇒ ChangeStruct (a, b, c) where
  type Δ(a, b, c) = (Δa, Δb, Δc)
  (a, b, c) ⊕ (da, db, dc) = (a ⊕ da, b ⊕ db, c ⊕ dc)
  oreplace (a₂, b₂, c₂) = (oreplace a₂, oreplace b₂, oreplace c₂)
```

Generalizing from pairs and triples, one can define similar instances for n-uples in general (say, for values of n up to some high threshold).

Validity and ⊕ on defunctionalized function changes

A function change df is valid for f if df has the same function code as f and if df's environment change is valid for f's environment:

$$DF\ d\rho\ c \triangleright F\ \rho_1\ c \hookrightarrow F\ \rho_2\ c : Fun\ \sigma\ \tau = d\rho \triangleright \rho_1 \hookrightarrow \rho_2 : env$$

where c is a function code of type $Code\ env\ \sigma\ \tau$, and where c's type binds the type variable env we use on the right-hand side.

Next, we implement ⊕ on function changes to match our definition of validity, as required. We only need $f \oplus df$ to give a result if df is a valid change for f. Hence, if the function code embedded in df does not match the one in f, we give an error.[3] However, our first attempt does not typecheck, since the typechecker does not know whether the environment and the environment change have compatible types.

```
instance ChangeStruct (Fun σ τ) where
  type Δ(Fun σ τ) = DFun σ τ
  F (env, c₁) ⊕ DF (denv, c₂) =
    if c₁ ≡ c₂ then
      F (env ⊕ denv) c₁
    else
      error "Invalid function change in oplus"
```

In particular, $env \oplus denv$ is reported as ill-typed, because we don't know that env and $denv$ have compatible types. Take $c_1 = Pair, c_2 = MapPair, f = F\ (env, Pair) :: \sigma \to \tau$ and $df =$

[3] We originally specified ⊕ as a total function to avoid formalizing partial functions, but as mentioned in Sec. 10.3, we do not rely on the behavior of ⊕ on invalid changes.

DF (*denv*, *MapPair*) :: $\Delta(\sigma \to \tau)$. Assume we evaluate $f \oplus df = F\ (env, Pair) \oplus DF\ (denv, MapPair)$: there, indeed, $env :: \sigma$ and $denv :: [\tau]$, so $env \oplus denv$ is not type-correct. Yet, evaluating $f \oplus df$ would *not* fail, because *MapPair* and *Pair* are different, $c_1 \equiv c_2$ will return false and $env \oplus denv$ won't be evaluated. But the typechecker does not know that.

Hence, we need an equality operation that produces a witness of type equality. We define the needed infrastructure with few lines of code. First, we need a GADT of witnesses of type equality; we can borrow from GHC's standard library its definition, which is just:

```
  -- From Data.Type.Equality
data τ1 :~: τ2 where
  Refl :: τ :~: τ
```

If x has type $\tau_1 :\sim: \tau_2$ and matches pattern *Refl*, then by standard GADT typing rules τ_1 and τ_2 are equal. Even if $\tau_1 :\sim: \tau_2$ has only constructor *Refl*, a match is necessary since x might be bottom. Readers familiar with type theory, Agda or Coq will recognize that :~: resembles Agda's propositional equality or Martin-Löf's identity types, even though it can only represents equality between types and not between values.

Next, we implement function *codeMatch* to compare codes. For equal codes, this operation produces a witness that their environment types match.[4] Using this operation, we can complete the above instance of *ChangeStruct* (*Fun* σ τ).

```
codeMatch :: Code env1 σ τ → Code env2 σ τ → Maybe (env1 :~: env2)
codeMatch AddOne AddOne = Just Refl
codeMatch Pair Pair = Just Refl
codeMatch MapPair MapPair = Just Refl
codeMatch _ _ = Nothing

instance ChangeStruct (Fun σ τ) where
  type Δ(Fun σ τ) = DFun σ τ
  F (c1, env) ⊕ DF (c2, denv) =
    case codeMatch c1 c2 of
      Just Refl → F (c1, env ⊕ denv)
      Nothing → error "Invalid function change in oplus"
```

Applying function changes

After defining environment changes, we define an incremental interpretation function *dapplyCode*. If c is the code for a function $f = \lambda x \to t$, as discussed, calling *applyCode c* computes the output of f from an environment *env* for f and an argument for x. Similarly, calling *dapplyCode c* computes the output of $\mathcal{D}[\![f]\!]$ from an environment *env* for f, an environment change *denv* valid for *env*, an argument for x and an argument change dx.

In our example, we have

```
dapplyCode :: Code env σ τ → env → Δenv → σ → Δσ → Δτ
dapplyCode   AddOne          ()     ()     x    dx =  dx
dapplyCode   Pair            x      dx     y    dy =  (dx, dy)
dapplyCode   MapPair         ys     dys    x    dx =
  dmapDF (F (Pair, x)) (DF (Pair, dx)) ys dys
```

[4] If a code is polymorphic in the environment type, it must take as argument a representation of its type argument, to be used to implement *codeMatch*. We represent type arguments at runtime via instances of *Typeable*, and omit standard details here.

On top of *dapplyCode* we can define *dapplyFun*, which functions as a a derivative for *applyFun* and allows applying function changes:

```
dapplyFun :: Fun σ τ → DFun σ τ → σ → Δσ → Δτ
dapplyFun (F (c1, env)) (DF (c2, denv)) x dx =
  case codeMatch c1 c2 of
    Just Refl → dapplyCode c1 env denv x dx
    Nothing → error "Invalid function change in dapplyFun"
```

However, we can also implement further accessors that inspect function changes. We can now finally detect if a change is nil. To this end, we first define a typeclass that allows testing changes to determine if they're nil:

```
class NilChangeStruct t ⇒ NilTestable t where
  isNil :: Δt → Bool
```

Now, a function change is nil only if the contained environment is nil.

```
instance NilTestable (Fun σ τ) where
  isNil :: DFun σ τ → Bool
  isNil (DF (denv, code)) = isNil denv
```

However, this definition of *isNil* only works given a typeclass instance for *NilTestable env*; we need to add this requirement as a constraint, but we cannot add it to *isNil*'s type signature since type variable *env* is not in scope there. Instead, we must add the constraint where we introduce it by existential quantification, just like the constraint *ChangeStruct env*. In particular, we can reuse the constraint *Env env*.

```
data DFun σ τ =
  ∀env. Env env ⇒
  DF (Code env σ τ, Δenv)
instance NilChangeStruct (Fun σ τ) where
  0_(F (code,env)) = DF (code, 0_env)
instance NilTestable (Fun σ τ) where
  isNil :: DFun σ τ → Bool
  isNil (DF (code, denv)) = isNil denv
```

We can then wrap a derivative via function *wrapDF* to return a nil change immediately if at runtime all input changes turn out to be nil. This was not possible in the setting described by Cai et al. [2014], because nil function changes could not be detected at runtime, only at compile time. To do so, we must produce directly a nil change for $v = \mathit{applyFun}\ f\ x$. To avoid computing v, we assume we can compute a nil change for v without access to v via operation *onil* and typeclass *OnilChangeStruct* (omitted here); argument *Proxy* is a constant required for purely technical reasons.

```
wrapDF :: OnilChangeStruct τ ⇒ Fun σ τ → DFun σ τ → σ → Δσ → Δτ
wrapDF f df x dx =
  if isNil df then
    onil Proxy   -- Change-equivalent to 0_(applyFun f x)
  else
    dapplyFun f df x dx
```

D.3 Defunctionalization and cache-transfer-style

We can combine the above ideas with cache-transfer-style (Chapter 17). We show the details quickly.

Combining the above with caching, we can use defunctionalization as described to implement the following interface for functions in caches. For extra generality, we use extension *ConstraintKinds* to allow instances to define the required typeclass constraints.

```
class FunOps k where
  type Dk k = (dk :: * → * → * → * → *) | dk → k
  type ApplyCtx k i o :: Constraint
  apply :: ApplyCtx k i o ⇒ k i o cache → i → (o, cache)
  type DApplyCtx k i o :: Constraint
  dApply :: DApplyCtx k i o ⇒
    Dk k i o cache1 cache2 → Δi → cache1 → (Δo, cache2)
  type DerivCtx k i o :: Constraint
  deriv :: DerivCtx k i o ⇒
    k i o cache → Dk k i o cache cache
  type FunUpdCtx k i o :: Constraint
  funUpdate :: FunUpdCtx k i o ⇒
    k i o cache1 → Dk k i o cache1 cache2 → k i o cache2
  isNilFun :: Dk k i o cache1 cache2 → Maybe (cache1 :∼: cache2)
updatedDeriv ::
  (FunOps k, FunUpdCtx k i o, DerivCtx k i o) ⇒
  k i o cache1 → Dk k i o cache1 cache2 → Dk k i o cache2 cache2
updatedDeriv f df = deriv (f `funUpdate` df)
```

Type constructor k defines the specific constructor for the function type. In this interface, the type of function changes $DK\ k\ i\ o\ cache_1\ cache_2$ represents functions (encoded by type constructor $Dk\ k$) with inputs of type i, outputs of type o, input cache type $cache_1$ and output cache type $cache_2$. Types $cache_1$ and $cache_2$ coincide for typical function changes, but can be different for replacement function changes, or more generally for function changes across functions with different implementations and cache types. Correspondingly, *dApply* supports applying such changes across closures with different implementations: unfortunately, unless the two implementations are similar, the cache content cannot be reused.

To implement this interface it is sufficient to define a type of codes that admits an instance of typeclass *Codelike*. Earlier definitions of *codeMatch*, *applyFun* and *dapplyFun*, adapted to cache-transfer style.

```
class Codelike code where
  codeMatch :: code env1 a1 r1 cache1 → code env2 a2 r2 cache2 →
    Maybe ((env1, cache1) :∼: (env2, cache2))
  applyCode :: code env a b cache → env → a → (b, cache)
  dapplyCode :: code env a b cache → Δenv → Δa → cache → (Δb, cache)
```

Typically, a defunctionalized program uses no first-class functions and has a single type of functions. Having a type class of function codes weakens that property. We can still use a single type of codes throughout our program; we can also use different types of codes for different parts of a program, without allowing for communications between those parts.

On top of the *Codelike* interface, we can define instances of interface *FunOps* and *ChangeStruct*. To demonstrate this, we show a complete implementation in Figs. D.3 and D.4. Similarly to $\oplus$, we

can implement ⊚ by comparing codes contained in function changes; this is not straightforward when using closure conversion as in Sec. 17.4.2, unless we store even more type representations.

We can detect nil changes at runtime even in cache-passing style. We can for instance define function *wrapDer1* which does something trickier than *wrapDF*: here we assume that *dg* is a derivative taking function change *df* as argument. Then, if *df* is nil, *dg df* must also be nil, so we can return a nil change directly, together with the input cache. The input cache has the required type because in this case, the type $cache_2$ of the desired output cache has matches type $cache_1$ of the input cache (because we have a nil change *df* across them): the return value of *isNilFun* witnesses this type equality.

```
wrapDer1 ::
  (FunOps k, OnilChangeStruct r') ⇒
  (Dk k i o cache₁ cache₂ → f cache₁ → (Δr', f cache₂)) →
  (Dk k i o cache₁ cache₂ → f cache₁ → (Δr', f cache₂))
wrapDer1 dg df c =
  case isNilFun df of
    Just Refl → (onil Proxy, c)
    Nothing → dg df c
```

We can also hide the difference between difference cache types by defining a uniform type of caches, *Cache code*. To hide caches, we can use a pair of a cache (of type *cache*) and a code for that cache type. When applying a function (change) to a cache, or when composing the function, we can compare the function code with the cache code.

In this code we have not shown support for replacement values for functions; we leave details to our implementation.

```
type RawFun a b code env cache = (code env a b cache, env)
type RawDFun a b code env cache = (code env a b cache, Δenv)

data Fun a b code = ∀env cache. Env env ⇒
  F (RawFun a b code env cache)
data DFun a b code = ∀env cache. Env env ⇒
  DF (RawDFun a b code env cache)

  -- This cache is not indexed by a and b
data Cache code = ∀a b env cache. C (code env a b cache) cache

  -- Wrapper
data FunW code a b cache where
  W :: Fun a b code → FunW code a b (Cache code)

data DFunW code a b cache₁ cache₂ where
  DW :: DFun a b code → DFunW code a b (Cache code) (Cache code)

derivFun :: Fun a b code → DFun a b code
derivFun (F (code, env)) = DF (code, 0_env)

oplusBase :: Codelike code ⇒ Fun a b code → DFun a b code → Fun a b code
oplusBase (F (c₁, env)) (DF (c₂, denv)) =
  case codeMatch c₁ c₂ of
    Just Refl →
      F (c₁, env ⊕ denv)
    _ → error "INVALID call to oplusBase!"

ocomposeBase :: Codelike code ⇒ DFun a b code → DFun a b code → DFun a b code
ocomposeBase (DF (c₁, denv₁)) (DF (c₂, denv₂)) =
  case codeMatch c₁ c₂ of
    Just Refl →
      DF (c₁, denv₁ ⊙ denv₂)
    _ → error "INVALID call to ocomposeBase!"

instance Codelike code ⇒ ChangeStruct (Fun a b code) where
  type Δ(Fun a b code) = DFun a b code
  (⊕) = oplusBase

instance Codelike code ⇒ NilChangeStruct (Fun a b code) where
  0_F (c,env) = DF (c, 0_env)

instance Codelike code ⇒ CompChangeStruct (Fun a b code) where
  df₁ ⊙ df₂ = ocomposeBase df₁ df₂

instance Codelike code ⇒ NilTestable (Fun a b code) where
  isNil (DF (c, env)) = isNil env
```

Figure D.3: Implementing change structures using *Codelike* instances.

```
applyRaw :: Codelike code ⇒ RawFun a b code env cache → a → (b, cache)
applyRaw (code, env) = applyCode code env

dapplyRaw :: Codelike code ⇒ RawDFun a b code env cache → Δa → cache → (Δb, cache)
dapplyRaw (code, denv) = dapplyCode code denv

applyFun :: Codelike code ⇒ Fun a b code → a → (b, Cache code)
applyFun (F f @ (code, env)) arg =
  (id *** C code) $ applyRaw f arg

dapplyFun :: Codelike code ⇒ DFun a b code → Δa → Cache code → (Δb, Cache code)
dapplyFun (DF (code₁, denv)) darg (C code₂ cache₁) =
  case codeMatch code₁ code₂ of
    Just Refl →
      (id *** C code₁) $ dapplyCode code₁ denv darg cache₁
    _ → error "INVALID call to dapplyFun!"
instance Codelike code ⇒ FunOps (FunW code) where
  type Dk (FunW code) = DFunW code
  apply (W f) = applyFun f
  dApply (DW df) = dapplyFun df
  deriv (W f) = DW (derivFun f)
  funUpdate (W f) (DW df) = W (f ⊕ df)
  isNilFun (DW df) =
    if isNil df then
      Just Refl
    else
      Nothing
```

Figure D.4: Implementing *FunOps* using *Codelike* instances.

Acknowledgments

No Man is an Island.

John Donne

Writing a PhD thesis is a significant challenge, both professionally and personally; in a sense, it's a solitary challenge, but in fact I've had so much help from so many, both in the research and in the personal growth that led me here. In fact, this page was the hardest to write, because I knew my words could not do justice to the task.

I'll start with my professional acknowledgments. I want to thank my supervisor Klaus Ostermann for encouraging me to do a PhD and for his years of patient and kind supervision; thanks also to Fritz Henglein for agreeing to review my thesis. Christian Kästner helped me become a better researcher and writer. Tillmann Rendel and Sebastian Erdweg sharpened my ability to articulate my arguments.

Many colleagues helped directly with work presented in this thesis. In particular, I'd like to thank again my colleagues Cai Yufei, Tillmann, Sebastian and Philipp Schuster, and my collaborators Lourdes Del Carmen Gonzàlez Huesca and Yann Règis-Gianas.

I also want to thank my PhD colleagues for their friendship, in particular Tillmann, Sebastian and Jonathan Brachthäuser, who helped make the challenges of research less solitary.

Most importantly, thanks to my late mother Rosalia, who sadly passed near the end of my PhD. I've struggled to thank her properly in this thesis, but I just can't; I'll just say I'm thankful for all she did for me, and that she's behind everything I am.

Finally, I'll thank my family and friends for all their support and entertainment: unfortunately, they are too many to list here, and I'd surely forget somebody who deserves to be listed, but they already know what I'd write here.

In sum, to all of you, *thank you*: I couldn't have done this without each of you.

Bibliography

[Citing pages are listed after each reference.]

Umut A Acar. *Self-Adjusting Computation.* PhD thesis, Carnegie Mellon University, 2005. [Pages 73, 135, 161, and 176.]

Umut A. Acar. Self-adjusting computation: (an overview). In *PEPM*, pages 1–6. ACM, 2009. [Pages 2, 128, 161, and 175.]

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *TOPLAS*, 28 (6):990–1034, November 2006. [Page 175.]

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328476. URL http://doi.acm.org/10.1145/1328438.1328476. [Pages 143, 146, 161, 196, 202, 203, 204, 206, 207, and 215.]

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *TOPLAS*, 32(1):3:1–3:53, November 2009. [Page 175.]

Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Turkoglu. Traceable data types for self-adjusting computation. In *PLDI*, pages 483–496. ACM, 2010. [Pages 161 and 175.]

Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded DSL for high performance big data processing. In *Int'l Workshop on End-to-end Management of Big Data (BigData)*, 2012. [Page 49.]

Agda Development Team. The Agda Wiki. http://wiki.portal.chalmers.se/agda/, 2013. Accessed on 2017-06-29. [Page 183.]

Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems*, pages 69–83. Springer, Berlin, Heidelberg, March 2006. doi: 10.1007/11693024_6. [Pages 143, 146, 161, 163, 195, 196, 203, 204, 206, 207, 212, 215, and 216.]

Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Yves Bertot and Viktor Vafeiadis, editors, *CPP 2017*. ACM, New York, NY, New York, January 2017. ISBN 978-1-4503-4705-1. [Page 188.]

Thorsten Altenkirch and Bernhard Reus. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, Lecture Notes in Computer Science, pages 453–468. Springer Berlin Heidelberg, September 1999. ISBN 978-3-540-66536-6 978-3-540-48168-3. doi: 10.1007/3-540-48168-0_32. [Page 188.]

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 666–679, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009866. URL http://doi.acm.org/10.1145/3009837.3009866. [Pages 188 and 203.]

Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *JFP*, 7:515–540, 1997. [Page 129.]

Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001. ISSN 0164-0925. doi: 10.1145/504709.504712. URL http://doi.acm.org/10.1145/504709.504712. [Page 161.]

Robert Atkey. The incremental λ-calculus and relational parametricity, 2015. URL http://bentnib.org/posts/2015-04-23-incremental-lambda-calculus-and-parametricity.html. Last accessed on 29 June 2017. [Pages 81, 126, 163, and 195.]

Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999. [Page 188.]

H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984. [Page 63.]

Henk P. Barendregt. *Lambda Calculi with Types*, pages 117–309. Oxford University Press, New York, 1992. [Pages 164, 166, and 171.]

Nick Benton, Chung-Kil Hur, Andrew J. Kennedy, and Conor McBride. Strongly Typed Term Representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, August 2012. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-011-9219-0. [Page 188.]

Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software*, FOSSACS'11/ETAPS'11, pages 108–122, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19804-5. [Pages 89, 163, 164, 165, 166, 168, 172, 173, and 179.]

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863592. URL http://doi.acm.org/10.1145/1863543.1863592. [Pages 166, 167, 172, and 173.]

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C#. In *OOPSLA*, pages 479–498. ACM, 2007. [Page 47.]

Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71. ACM, 1986. [Pages v, vii, 2, 161, and 176.]

Maximilian C. Bolingbroke and Simon L. Peyton Jones. Types are calling conventions. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: 10.1145/1596638.1596640. [Page 160.]

K.J. Brown, A.K. Sujeeth, Hyouk Joong Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100, 2011. doi: 10.1109/PACT.2011.15. [Page 78.]

Yufei Cai. PhD thesis, University of Tübingen, 2017. In preparation. [Pages 81 and 126.]

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages — Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594304. URL http://doi.acm.org/10.1145/2594291.2594304. [Pages 2, 3, 59, 66, 84, 92, 93, 107, 117, 121, 124, 125, 135, 136, 137, 138, 141, 145, 146, 154, 159, 177, 179, 184, 186, 192, and 225.]

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *JFP*, 19:509–543, 2009. [Page 31.]

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375. ACM, 2010. [Pages 10, 20, and 47.]

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *ICFP*, pages 129–141. ACM, 2011. [Page 175.]

Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 227–240, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628150. URL http://doi.acm.org/10.1145/2628136.2628150. [Page 161.]

James Cheney, Sam Lindley, and Philip Wadler. A Practical Theory of Language-integrated Query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 403–416, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500586. [Page 176.]

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411226. URL http://doi.acm.org/10.1145/1411204.1411226. [Page 188.]

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. A type theory for incremental computational complexity with control flow changes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 132–145, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951950. [Page 161.]

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001. ISBN 0-262-03293-7, 9780262032933. [Page 73.]

Oren Eini. The pain of implementing LINQ providers. *Commun. ACM*, 54(8):55–61, 2011. [Page 47.]

Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997. [Page 176.]

Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *JFP*, 13(2):455–481, 2003. [Pages 20 and 48.]

Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C$^\sharp$ generics. In *ECOOP*, pages 279–303. Springer-Verlag, 2006. [Page 4.]

Burak Emir, Qin Ma, and Martin Odersky. Translation correctness for first-order object-oriented pattern matching. In *Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, pages 54–70, Berlin, Heidelberg, 2007a. Springer-Verlag. ISBN 3-540-76636-7, 978-3-540-76636-0. URL http://dl.acm.org/citation.cfm?id=1784774.1784782. [Page 4.]

Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP*, pages 273–298. Springer-Verlag, 2007b. [Pages 4, 32, 44, and 45.]

Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1536-4. doi: 10.1145/2427048.2427055. URL http://doi.acm.org/10.1145/2427048.2427055. [Pages 4 and 189.]

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-Avoiding and Hygienic Program Transformations. In *ECOOP 2014 – Object-Oriented Programming*, pages 489–514. Springer, Berlin, Heidelberg, July 2014. doi: 10.1007/978-3-662-44202-9_20. [Page 93.]

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364508. [Page 78.]

Leonidas Fegaras. Optimizing queries with object updates. *Journal of Intelligent Information Systems*, 12:219–242, 1999. ISSN 0925-9902. URL http://dx.doi.org/10.1023/A:1008757010516. 10.1023/A:1008757010516. [Page 71.]

Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 47–58, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223789. URL http://doi.acm.org/10.1145/223784.223789. [Page 71.]

Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Systems (TODS)*, 25:457–516, 2000. [Pages 11, 21, and 48.]

Denis Firsov and Wolfgang Jeltsch. Purely functional incremental computing. In Fernando Castor and Yu David Liu, editors, *Programming Languages*, number 9889 in Lecture Notes in Computer Science, pages 62–77. Springer International Publishing, September 2016. ISBN 978-3-319-45278-4 978-3-319-45279-1. doi: 10.1007/978-3-319-45279-1_5. [Pages 70, 75, 156, and 160.]

Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986. [Pages xvii, 39, and 41.]

Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative Inquiry*, 12(2): 219–245, 2006. [Page 23.]

Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending Scala with database query capability. *Journal of Object Technology*, 9(4):45–68, 2010. [Page 48.]

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76. ACM, 2007. [Pages 40, 41, and 132.]

Paolo G. Giarrusso. Open GADTs and declaration-site variance: A problem statement. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 5:1–5:4, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2064-1. doi: 10.1145/2489837.2489842. URL http://doi.acm.org/10.1145/2489837.2489842. [Page 4.]

Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify your collection queries for modularity and speed! *CoRR*, abs/1210.6284, 2012. URL http://arxiv.org/abs/1210.6284. [Page 20.]

Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify your collection queries for modularity and speed! In *AOSD*, pages 1–12. ACM, 2013. [Pages 2 and 3.]

Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental λ-calculus in cache-transfer style. In Luís Caires, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 553–580. Springer International Publishing, 2019. ISBN 978-3-030-17184-1. [Pages 3 and 215.]

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232. ACM, 1993. [Page 9.]

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press Cambridge, 1989. [Page 164.]

Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc Scholl. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, volume 1341 of *LNCS*, pages 52–66. Springer, 1997. [Pages 57, 127, 129, 131, and 176.]

T. Grust and M.H. Scholl. Monoid comprehensions as a target for the translation of oql. In *Workshop on Performance Enhancement in Object Bases, Schloss Dagstuhl*. Citeseer, 1996a. doi: 10.1.1.36.4925,10.1.1.36.4481(abstract). [Page 71.]

Torsten Grust. *Comprehending queries*. PhD thesis, University of Konstanz, 1999. [Pages 32 and 48.]

Torsten Grust and Marc H. Scholl. Translating OQL into monoid comprehensions: Stuck with nested loops? Technical Report 3a/1996, University of Konstanz, Department of Mathematics and Computer Science, Database Research Group, September 1996b. [Page 32.]

Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *Journal of Intelligent Information Systems*, 12:191–218, 1999. [Page 21.]

Torsten Grust and Alexander Ulrich. First-class functions for first-order database engines. In Todd J. Green and Alan Schmitt, editors, *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy.*, 2013. URL http://arxiv.org/abs/1308.0158. [Page 176.]

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: database-supported program execution. In *Proc. Int'l SIGMOD Conf. on Management of Data (SIGMOD)*, pages 1063–1066. ACM, 2009. [Pages 48 and 176.]

Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. Functions Are Data Too: Defunctionalization for PL/SQL. *Proc. VLDB Endow.*, 6(12):1214–1217, August 2013. ISSN 2150-8097. doi: 10.14778/2536274.2536279. [Page 176.]

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: problems, techniques, and applications. In Ashish Gupta and Iderpal Singh Mumick, editors, *Materialized views*, pages 145–157. MIT Press, 1999. [Pages v, vii, 2, 128, 160, 175, and 176.]

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest*: Scalable source code queries with Datalog. In *ECOOP*, pages 2–27. Springer, 2006. [Page 49.]

Matthew A. Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *OOPSLA*, pages 753–772. ACM, 2011. [Page 175.]

Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594324. [Page 161.]

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 748–766, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814305. URL http://doi.acm.org/10.1145/2814270.2814305. [Page 161.]

Matthew A. Hammer, Joshua Dunfield, Dimitrios J. Economou, and Monal Narasimhamurthy. Typed Adapton: refinement types for incremental computations with precise names. *arXiv:1610.00097 [cs]*, October 2016. [Page 161.]

Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14(1):71–84, July 1992. ISSN 0747-7171. doi: 10.1016/0747-7171(92)90026-Z. [Page 118.]

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016. ISBN 9781107150300. [Page 163.]

Kyle Headley and Matthew A. Hammer. The Random Access Zipper: Simple, Purely-Functional Sequences. *arXiv:1608.06009 [cs]*, August 2016. [Page 73.]

Fritz Henglein. Optimizing Relational Algebra Operations Using Generic Equivalence Discriminators and Lazy Products. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 73–82, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1. doi: 10.1145/1706356.1706372. [Page 48.]

Fritz Henglein and Ken Friis Larsen. Generic Multiset Programming for Language-integrated Querying. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 49–60, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863503. [Page 48.]

R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–218, 2006. [Pages 73 and 75.]

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *GPCE*, pages 137–148. ACM, 2008. [Page 44.]

Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *Types for Proofs and Programs*, volume 1158 of *LNCS*, pages 153–164. Springer-Verlag, 1996. [Page 184.]

David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004. [Pages 10, 37, and 49.]

Lourdes del Carmen Gonzalez Huesca. *Incrementality and Effect Simulation in the Simply Typed Lambda Calculus*. PhD thesis, Universite Paris Diderot-Paris VII, November 2015. [Pages 59 and 177.]

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: Concealing the deep embedding of dsls. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 73–82, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3161-6. doi: 10.1145/2658761.2658771. URL http://doi.acm.org/10.1145/2658761.2658771. [Page 49.]

Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann. Partial preprocessing C code for variability analysis. In *Proceedings of the Fifth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 137–140. ACM, January 2011a. ISBN 978-1-4503-0570-9. URL http://www.informatik.uni-marburg.de/~kaestner/vamos11.pdf. [Page 4.]

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, October 2011b. [Page 4.]

Chantal Keller and Thorsten Altenkirch. Hereditary Substitutions for Simple Types, Formalized. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0255-5. doi: 10.1145/1863597.1863601. [Page 188.]

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. [Page 9.]

Edward Kmett. Mirrored lenses. Last accessed on 29 June 2017, 2012. URL http://comonad.com/reader/2012/mirrored-lenses/. [Page 169.]

Christoph Koch. Incremental query evaluation in a ring of databases. In *Symp. Principles of Database Systems (PODS)*, pages 87–98. ACM, 2010. [Pages 57, 161, and 176.]

Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014. ISSN 1066-8888. doi: 10.1007/s00778-013-0348-4. URL http://dx.doi.org/10.1007/s00778-013-0348-4. [Pages 153, 154, 158, 161, and 176.]

Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 75–90, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4191-2. doi: 10.1145/2902251.2902286. [Pages 2, 57, 159, 161, and 177.]

Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013)*, volume 23 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 432–451, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-60-6. doi: http://dx.doi.org/10.4230/LIPIcs.CSL.2013.432. [Page 125.]

Ralf Lämmel. Google's MapReduce programming model — revisited. *Sci. Comput. Program.*, 68(3): 208–237, October 2007. [Pages 67 and 129.]

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122. ACM, 1999. [Page 48.]

Yanhong A. Liu. Efficiency by incrementalization: an introduction. *HOSC*, 13(4):289–313, 2000. [Pages 2, 56, 160, 178, and 180.]

Yanhong A. Liu and Tim Teitelbaum. Caching intermediate results for program improvement. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '95, pages 190–201, New York, NY, USA, 1995. ACM. ISBN 0-89791-720-0. doi: 10.1145/215465.215590. URL http://doi.acm.org/10.1145/215465.215590. [Pages v, vii, 135, and 136.]

Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731. Springer-Verlag, 2013. [Pages 43 and 176.]

Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006. ISSN 1469-7653. doi: 10.1017/S0956796806005995. [Page 160.]

Conor McBride. The derivative of a regular type is its type of one-hole contexts. Last accessed on 29 June 2017, 2001. URL http://www.strictlypositive.org/diff.pdf. [Page 78.]

Conor McBride. Outrageous but Meaningful Coincidences: Dependent Type-safe Syntax and Evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0251-7. doi: 10.1145/1863495.1863497. [Page 188.]

Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling objects, relations and XML in the .NET framework. In *Proc. Int'l SIGMOD Conf. on Management of Data (SIGMOD)*, page 706. ACM, 2006. [Page 47.]

John C. Mitchell. *Foundations of programming languages.* MIT Press, 1996. [Page 118.]

Adriaan Moors. Type constructor polymorphism, 2010. URL http://adriaanm.github.com/research/2010/10/06/new-in-scala-2.8-type-constructor-inference/. Last accessed on 2012-04-11. [Page 45.]

Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proc. Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '12, pages 117–120. ACM, 2012. [Page 44.]

Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997. ISBN 1-55860-320-4. [Page 21.]

Russell O'Connor. Polymorphic update with van Laarhoven lenses. Last accessed on 29 June 2017, 2012. URL http://r6.ca/blog/20120623T104901Z.html. [Page 169.]

M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 4, pages 427–451, 2009. [Pages 20, 23, 30, and 48.]

Martin Odersky. Pimp my library, 2006. URL http://www.artima.com/weblogs/viewpost.jsp?thread=179766. Last accessed on 10 April 2012. [Page 25.]

Martin Odersky. The Scala language specification version 2.9, 2011. [Page 25.]

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima Inc, 2nd edition, 2011. [Pages 10, 23, 25, 45, and 46.]

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, OOPSLA '10, pages 341–360. ACM, 2010. [Page 30.]

Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2011. URL http://www.informatik.uni-marburg.de/~kaestner/ecoop11.pdf. [Page 3.]

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*, pages 589–615, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_23. [Pages 188 and 203.]

Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3): 402–454, July 1982. [Pages v, vii, 2, 57, 161, and 176.]

Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *JFP*, 12 (4-5):393–434, 2002. [Pages 9 and 32.]

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI*, pages 199–208. ACM, 1988. [Pages 20 and 28.]

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. ACM Conf. on LISP and Functional Programming*, LFP '88, pages 153–163. ACM, 1988. [Page 45.]

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1st edition, 2002. [Pages 33, 183, 185, 186, and 203.]

François Pottier and Nadji Gauthier. Polymorphic Typed Defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 89–98, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964009. [Page 219.]

G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *POPL*, pages 502–510. ACM, 1993. [Pages 55, 160, and 175.]

Tiark Rompf and Nada Amin. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 2–9, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784760. [Page 1.]

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010. [Pages 4, 20, 21, 22, 25, 44, 49, 78, and 127.]

Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented DSLs. In *DSL*, pages 93–117, 2011. [Pages 44, 46, and 49.]

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *POPL*, pages 497–510. ACM, 2013. [Pages 43, 48, and 49.]

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 89–102, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9. doi: 10.1145/1708016.1708028. [Page 154.]

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3-4):289–360, 1993. [Page 138.]

Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *AOSD*, pages 37–48. ACM, 2013. [Pages 1 and 55.]

Gabriel Scherer and Didier Rémy. GADTs meet subtyping. In *ESOP*, pages 554–573. Springer-Verlag, 2013. [Page 4.]

Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 3–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1860-0. doi: 10.1145/2428116.2428120. [Page 192.]

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. Modular, higher-order cardinality analysis in theory and practice. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 335–347, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535861. [Page 160.]

Rohin Shah and Rastislav Bodik. Automated incrementalization through synthesis. Presented at *IC 2017, First Workshop on Incremental Computing*; 2-page abstract available, 2017. URL http://pldi17.sigplan.org/event/ic-2017-papers-automated-incrementalization-through-synthesis. [Page 56.]

Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, Los Alamitos, CA, August 2011. IEEE Computer Society. URL http://www.informatik.uni-marburg.de/~kaestner/SPLC11_nfp.pdf. to appear; submitted 27 Feb 2010, accepted 21 Apr 2011. [Page 4.]

Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013. [Page 4.]

Daniel Spiewak and Tian Zhao. ScalaQL: Language-integrated database queries for Scala. In *Proc. Conf. Software Language Engineering (SLE)*, 2009. [Page 48.]

R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65(2):85–97, May 1985. ISSN 0019-9958. doi: 10.1016/S0019-9958(85)80001-2. [Page 89.]

Guy L. Steele, Jr. Organizing Functional Code for Parallel Execution or, Foldl and Foldr Considered Slightly Harmful. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596551. [Page 43.]

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 1150–1160. VLDB Endowment, 2007. [Pages 1 and 9.]

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance dsl implementation from a declarative specification. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, GPCE '13, pages 145–154, New York, NY, USA, 2013a. ACM. ISBN 978-1-4503-2373-4. doi: 10.1145/2517208.2517220. URL http://doi.acm.org/10.1145/2517208.2517220. [Page 49.]

Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, pages 52–78. Springer-Verlag, 2013b. [Pages 43 and 49.]

R. S. Sundaresh and Paul Hudak. A theory of incremental computation and its application. In *POPL*, pages 1–13. ACM, 1991. [Page 176.]

L. S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for Pure Type Systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, number 806 in Lecture Notes in Computer Science, pages 19–61. Springer Berlin Heidelberg, January 1994. ISBN 978-3-540-58085-0 978-3-540-48440-0. [Page 172.]

Jan Vitek and Tomas Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proc. Int'l Conf. Embedded Software (EMSOFT)*, pages 33–38. ACM, 2011. [Page 36.]

Philip Wadler. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 978-0-89791-328-7. doi: 10.1145/99370.99404. [Page 89.]

Philip Wadler. The Girard–Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.042. [Page 163.]

Patrycja Węgrzynowicz and Krzysztof Stencel. The good, the bad, and the ugly: three ways to use a semantic code query system. In *OOPSLA*, pages 821–822. ACM, 2009. [Page 49.]

Darren Willis, David Pearce, and James Noble. Efficient object querying for Java. In *ECOOP*, pages 28–49. Springer, 2006. [Page 48.]

Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the Java Query Language. In *OOPSLA*, pages 1–18. ACM, 2008. [Pages 48 and 176.]

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. Conf. Operating systems design and implementation*, OSDI'08, pages 1–14. USENIX Association, 2008. [Page 47.]