A flow-analysis framework for realistic Scheme programs

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Eric Jean Knauel
aus Buchholz in der Nordheide

Tübingen
2008

Tag der mündlichen Qualifikation:	7. Mai 2008
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Schroeder-Heister

Abstract

It is possible to scale control-flow analyses for higher-order languages to complete, fully-fledged programming languages and consequently compute the flow analysis of realistic programs. This dissertation gives a formal specification of a universal flow-analysis framework for the functional programming languages Scheme and PreScheme that covers all aspects and features of these languages. Moreover, the dissertation proposes new implementation strategies and techniques that have been developed and tested in context of an implementation for Scheme 48. These techniques yield an efficient implementation that enables analysis of realistic programs.

Zusammenfassung

Es ist möglich, Kontrollflussanalysen für Sprachen höherer Ordnung auf realistische Programme anzuwenden. Diese Dissertation spezifiziert eine universell verwendbare Flussanalyse für die funktionalen Programmiersprachen Scheme und PreScheme, die alle Aspekte und Fähigkeiten dieser Sprachen abdeckt. Für die praktische Umsetzung dieser Analyse werden neuartige Implementierungsstrategien und -techniken vorgestellt, die im Rahmen einer Implementierung für Scheme 48 entwickelt und erprobt wurden. Diese Techniken führen zu einer effizienten Implementierung, welche die Analyse realistischer Programme erlaubt.

Contents

1 Introduction **9**
- 1.1 Implementation Scalability 9
- 1.2 Executable semantic model 12
- 1.3 Setting the scene 13
- 1.4 Contributions 14
- 1.5 Structure of this dissertation 14

2 Intermediate Language **17**
- 2.1 Syntax 17
- 2.2 Printing CPS Programs 22
- 2.3 Semantics 24
- 2.4 Complete programs 30
  - 2.4.1 Complete PreScheme programs 30
  - 2.4.2 Complete Scheme programs 31
  - 2.4.3 Treating complete programs 31
  - 2.4.4 Initial state 32

3 Flow analysis of Programs **33**
- 3.1 Control flow 33
- 3.2 Using control-flow information for optimization 35
- 3.3 Control-flow analysis for higher-order languages 36

4 Analysis Framework **39**
- 4.1 Semantic domains 39
- 4.2 State transition 43
- 4.3 Semantic correctness 48
- 4.4 Language-specific abstractions 52
  - 4.4.1 Abstracting PreScheme values 53
  - 4.4.2 Abstracting Scheme values 60
- 4.5 Precision and $\widehat{\textbf{Time}}$ 73
  - 4.5.1 *k*-CFA time abstractions 74
- 4.6 Flow analysis examples 75

5 Executable Semantic Model **83**
- 5.1 Example trace 84
- 5.2 Implementation with PLT Redex 86
  - 5.2.1 Grammar definition 86
  - 5.2.2 Metafunctions 90

5.2.3 Reduction relations . . . 100
5.2.4 Generating traces . . . 102

6 Implementation 105
6.1 Abstract syntax . . . 105
6.2 Semantic domains . . . 106
6.2.1 Representing time . . . 106
6.2.2 Representing the binding environment . . . 106
6.2.3 Representing variable environments . . . 107
6.2.4 Representing abstract values . . . 107
6.2.5 Implementing the visited set . . . 112
6.3 Description of flow information . . . 114
6.4 Analysis with Garbage Collection . . . 116
6.4.1 Semantics with global garbage collection . . . 117
6.4.2 Abstract values with GC marks . . . 122
6.4.3 Garbage Collection and variable environment . . . 124
6.4.4 Splitting the root set . . . 127
6.5 Experimental results . . . 127
6.6 Testing and Debugging . . . 131

7 Conclusion 133
7.1 Review . . . 133
7.2 Future work . . . 134

A Notation 137

B Semantic Correctness 139

C PreScheme Primops 149
C.1 Primops with one return value . . . 149
C.2 Primops for compound values . . . 151
C.3 Primops with multiple return values . . . 153

D Scheme Primops 155
D.1 Arithmetic Primops . . . 157
D.2 Primops for compound values . . . 158
D.3 Procedures with rest list argument . . . 161
D.4 Multiple return values . . . 162

Acknowledgments

First, I thank my advisor Professor Herbert Klaeren. Professor Klaeren provided a pleasant work environment and allowed me great latitude in my choice of research activities, and agreed with this dissertation project. I thank Professor Peter Schroeder-Heister who immediately agreed to co-review this dissertation.

I am greatly indebted to Mike Sperber. Mike was the first to direct my attention to research in compiler construction and especially to flow analyses. He wakened my curiosity for these subjects. I am deeply thankful that Mike shared his extensive knowledge with me, and the time and effort he spent reading the various drafts of this dissertation. Mike's profound remarks, numerous corrections and suggestions — the remaining mistakes are mine — really helped me improve this work.

My colleague Marcus Crestani deserves special thanks. Marcus not only carefully proof-read this dissertation and suggested valuable improvements, but also relieved me of parts of my teaching workload during the last semester.

I am grateful to Matthew Might and Olin Shivers for patiently answering all my e-mails on flow analyses, garbage collection for flow analyses, and frame strings. Matthew Might also gave me access to his ΔCFA and ΓCFA implementations. Matthew and Olin also shared draft versions of their latest research results with me. For all of this, I am grateful.

I thank Robby Findler for answering my questions on PLT Redex — using this tool to model the semantics ranks among the most pleasant parts of this work.

Martin Gasbichler is an expert on Scheme 48 and I thank him for patiently answering all my questions on the byte code, native code, and PreScheme compilers, the virtual machine, and other aspects of this complex system. Especially, I thank Martin for helping me develop the id tables for Scheme 48. My colleague Holger Gast was always available to help me with typesetting problems — I would probably still be struggling with strange TeX problems without his help.

Frank Sühl and Thomas Rasch, both close friends of mine, provided many helpful comments and encouragement — for both I am grateful.

I am very grateful for the support from my parents Hermann and Marie-Louise. Marie-Louise proof-read the whole work and suggested lots of linguistic improvements. Both supported and encouraged me tireless from day one of my studies till the end of this dissertation project.

Finally, I thank my wife Gertraud for her love, support, and patience during the months I suffered from something she identified as the "compiling syndrome" — a state of mind, characterized by confusion and mental absence, caused by long debugging sessions of flow analyses.

Chapter 1

Introduction

It is possible to scale control-flow analyses for higher-order languages to complete, fully-fledged programming languages and consequently compute the flow analysis of realistic programs.

Only few compilers for functional languages employ a flow analysis to gain information on the run-time behavior of a program and subsequently optimize the program. This failure of flow analysis for functional languages in practice is paradoxical, as a rich amount of technical literature proposes new techniques for computing analyses, develops more precise abstractions, and finds new applications for analyses. The following observations help understanding the reasons for this paradox:

- Technical literature considers minimalist toy languages, leaving open many questions that arise when adapting an analysis for a real programming language.
- There are very few reports on experience with implementation techniques for flow analyses. Most literature only reports on prototype and proof-of-concept implementations.
- Only few descriptions of flow analyses show how they scale to realistic programs.

This dissertation aims at eliminating these shortcomings.

This introduction describes the problems that arise when scaling to a complete language (Section 1.1). Section 1.2 motivates the need for an executable semantic model. Section 1.3 describes the compiler architecture that is the basis for my flow analysis. A summary of the contributions (Section 1.4) and an overview of dissertation (Section 1.5) follow.

1.1 Implementation Scalability

Research literature in the area of flow analyses for higher-order languages presents new ideas as formal specifications with corresponding soundness proofs, accompanied by measurements supposed to show the relevance and usefulness of the

proposed ideas. It is striking that almost all measurements take place in proof-of-concept implementations, or at best, prototype implementations. These implementations have in common that they feature only a minimal subset of language constructs and are only applied to small programs.

However, there are a few notable exceptions:

- The Bigloo Scheme compiler [*Serrano*, 2004] uses a special-purpose flow analysis to optimize closure allocation [*Serrano*, 1995].
- The MLton compiler for Standard ML uses a flow analysis to perform a closure conversion that translates the code to a first-order intermediate language [*Cejtin et al.*, 2000]. Subsequently, MLton carries out optimizations on the first-order language.
- In the past, the Chez Scheme compiler used a flow analysis to improve the inlining optimization [*Jagannathan and Wright*, 1996; *Ashley*, 1997]. The flow-directed inlining algorithm, however, was replaced in favor of a special inlining algorithm that does not require a flow analysis [*Waddell and Dybvig*, 1997] — the developers considered a flow analysis slow and impractical [*Dybvig*, 2006].

Each of these exceptions use a flow analysis geared towards a single and specific purpose. This dissertation, however, considers a flow analysis that produces results that are useful for more than one application: It computes the control flow of the program and information on the values used in a program. The analysis is more general than the analyses described above as its results can be used to drive more than one optimization. For example, the results may be used for an inlining optimization as well as for the elimination of run-time type checks. Additionally, this dissertation gives a formal specification that covers the complete PreScheme and Scheme programming languages while the cited publications only consider the aspects of the language that are relevant for the specific purpose of the analysis.

Defining a flow analysis for a full programming language is different from specifying an analysis for a small and artificial language. There are three challenges for scaling a flow analysis: Adding support for complex language features, integration with a compiler, and developing efficient implementation techniques.

Consider language features first. The challenge in supporting a complete language lies in the interaction and dependencies of the features: It is often not possible to treat the features independently. Procedure calls with variable arity, for example, may allocate heap objects and thus the analysis needs to model heap objects. The abstraction functions for heap objects, on the other hand, rely on calls to distinguish references. For a precise model of compound heap objects a specific representation of integers in the analysis is indispensable, and so on.

Consequently, the analysis specified in this dissertation considers *all* features of Scheme (and PreScheme) in concert. To my knowledge, this coherent and complete specification of a general-purpose flow analysis for Scheme is novel. In addition to the core features the specification also covers the following features:

- Procedure calls with variable arity and a precise abstraction of the rest list.

- Procedures that return multiple values at once.
- A precise abstraction for records, pairs, and other compound heap objects. The analysis is able to track the values for each field separately.
- An abstraction for vectors.
- Precise abstractions for all simple value types of Scheme, such as booleans, characters, and so on.
- An abstraction for Scheme numbers that includes all three properties: type, value, and exactness.
- Support for most primitive operations of the Scheme 48 system and the PreScheme compiler.

To develop, trace, and validate the semantics of the flow analysis an executable model of the semantics is helpful (see Section 1.2).

The second challenge is the integration into a real compiler. Technical literature on flow analyses assumes that the complete program, including all dependencies, is available as source code. This assumption, however, is not realistic. The following problems must be addressed in context of a real compiler:

Libraries and run-time system Realistic programs use libraries. Libraries often contain low-level functionality that is deeply interweaved with the run-time system. The source code for such libraries is not always available. For example, the library code may have the form of compiled byte code, distributed as a heap image, or the code comes as a shared library containing machine code or foreign code. Only the program is subject to an optimization as the compiler simply links standard library code and the run-time system to the residual program. That is, it is not useful to compute a flow analysis for library code or the run-time system. As a consequence, the flow analysis has to know how calls into library code affect the flow analysis of the program without analyzing the library itself.

External code Besides standard library code, realistic programs often involve external code written in a different programming language. The problem is similar to the library problem just sketched. Library code can be deployed along with additional information for the flow analysis. External code, however, is written by the programmer. Therefore, the programmer needs a way to specify the behavior of the external code in the analysis.

This establishes why the description of a flow analysis for a complete programming language can only the viewed in context of a real compiler. In this dissertation, I specify an analysis for two complete programming languages: Scheme and PreScheme. The analysis operates as part of the transformational compiler shipped with Scheme 48 [*Kelsey and Rees*, 1995].

Only few reports of flow analyses exist that investigate how well the various implementation techniques scale to larger programs. The performance of the analysis, however, is the main challenge. Typically, new ideas in the field of flow analyses employ — as just discussed — toy languages and prototype implementations to demonstrate the usefulness and relevance. Investigations that get over this initial stage and consider realistic programs are rare. The few that

do exist do not report on implementation techniques. Consequently, this work focuses on the practical aspects of flow analysis, specifically those concerning performance and scalability.

My implementation deals with realistic programs: It is capable of analyzing the complete source code of the Scheme 48 virtual machine and substantial Scheme programs. Naive implementations take hours, even on simple programs. Sophisticated implementation techniques were necessary to lift the implementation from toy examples to the standard of realistic programs and compute the analysis in reasonable time. To my knowledge, the systematic investigation of the practical aspects of a flow analysis implementation reported in this dissertation is new.

The following ideas improve the analysis time considerably and identify two important contributions of this dissertation:

- A systematic investigation and design of the representation for semantic domains.
- A novel global garbage collection technique for flow analyses.

Consider the representation of semantic domains first. Choosing a representation that saves time and space is crucial for improving analysis time. Here, the performance of the approximation relation has a key role. This relation determines whether an abstract entity of the analysis is a proper simulation of another abstract entity. Computing this relation for complex semantic domains such as function domains is costly and occurs frequently. Consequently, the implementation uses representations for semantic domains that are geared towards a fast implementation of this relation. These representations allow to short-circuit checks, cache the results of expensive checks, and enable the definition of hash functions that reduce the number of necessary checks.

Recent research introduced the idea of garbage collection to flow-analyses that leads to improvements in precision and performance of an analysis [*Might and Shivers*, 2006]. For realistic programs, however, garbage collection can be costly. The global garbage collection technique in my implementation is a variation of Might's and Shivers's algorithm and considers the complete state of an analysis. This global garbage collector allows the analysis to use a single and global variable environment, which has two advantages: The analysis saves space and it becomes possible to replace the expensive approximation-relation check for the variable environment with a quick check that uses timestamps. Additionally, the collector uses the classical idea of a marking bit that distinguishes seen from unseen values during the traversal of the heap. This work shows how this classic idea of garbage collection can be applied to garbage collectors for flow analyses.

1.2 Executable semantic model

The formal specification of a flow analysis may suffer from the following intrinsic problems: First, the specification does not correctly capture the behavior of a program. Second, the implementation does not correctly match the specification. This dissertation addresses these problems by developing an executable model of the semantics. The executable model is based on PLT Redex [*Matthews*

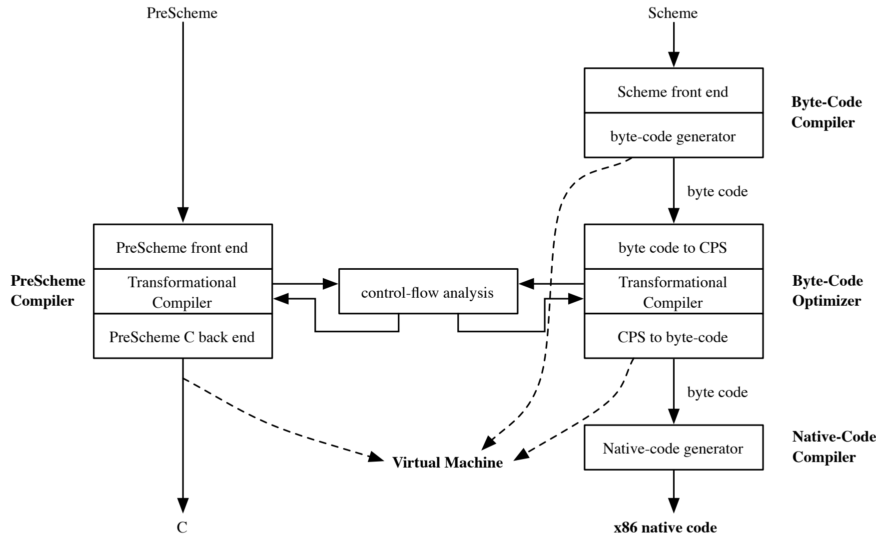

Figure 1.1: Scheme 48 native-code compiler [*Kelsey and Rees*, 1995; *Gasbichler and Sperber*, 2007] and the flow analysis

et al., 2004], a tool that allows to specify formal semantics in form of a term rewrite system and subsequently apply the reductions specified in the semantics.

The model describes the flow analysis as a term rewrite system. The analysis semantics is accompanied by a relation that demonstrates the correctness of the analysis by relating the analysis results with their counterpart in a concrete program run. For a given program that serves as a test case, the model computes all reduction steps in the analysis semantics and all steps in the standard PreScheme semantics. With the correctness relation at hand, the model can check and visualize the correctness of the analysis results. To my knowledge, the approach to use an executable model to test the correctness of an analysis and inspect its mode of operation is novel.

1.3 Setting the scene

The flow analysis specified in this dissertation operates on the intermediate language of the transformation compiler [*Kelsey*, 1989]. The transformational compiler is used for compiling the Scheme 48 virtual machine to C [*Kelsey and Rees*, 1995] and as the optimizer for byte code in the native-code compiler [*Gasbichler and Sperber*, 2007]. Figure 1.1 illustrates the compiler architecture of Scheme 48. Consider the PreScheme compiler [*Kelsey*, 1997] first: This compiler is written in Scheme and translates PreScheme code to portable C code. The PreScheme compiler converts PreScheme code to a *continuation passing style (CPS)* intermediate language, checks types, and uses the transformational compiler to remove any dependencies on the intermediate language semantics that C code cannot implement directly. The result is CPS code with properties that make it simple for the back end to generate C code.

The right-hand side of Figure 1.1 shows the compiler chain for the native-code compiler. By default, Scheme 48 compiles Scheme programs to byte code and runs the byte code on the virtual machine. The optional native-code compiler translates byte-code procedures to native code for Intel's 386 processor family.

Optionally, the user may optimize the byte code by using the byte-code optimizer. The byte-code optimizer turns recursive calls into jumps and carries out constant folding and various other optimizations implemented by the simplifier of the transformational compiler. The optimizer may be viewed as a front end and back end to the transformational compiler: The front end converts byte code to CPS intermediate language and the back end reverses this transformation. Typically, a user would invoke both, the optimizer and the native-code compiler, to produce efficient native code.

To this compiler architecture I have added the flow analysis specified in this dissertation. The analysis operates on the transformational compiler's intermediate language and handles both Scheme and PreScheme code. The language-specific abstraction functions and definition of primitive operations reside in separate modules and share a common interface.

1.4 Contributions

In summary, the main contributions of this dissertation are the following:

- A formal definition of the syntax and semantics of the transformational compiler's intermediate language.
- A specification for a universal flow analysis that covers all features of the PreScheme and Scheme programming languages.
- An executable model of the flow analysis semantics.
- Implementation techniques for the analysis framework that scale and enable the analysis of realistic programs.
- A novel garbage-collection technique for flow analyses geared towards performance.

1.5 Structure of this dissertation

This dissertation has the following structure: Chapter 2 defines the syntax and semantics of the transformational compiler's intermediate language. Chapter 3 contains a brief overview on control flow and and data-flow analyses and their applications. Chapter 4 defines the flow analysis framework, formulates a correctness criterion, specifies the semantic domains, abstract functions and primitive operations for PreScheme and Scheme. Chapter 5 describes the executable model for the semantics. Chapter 6 describes the implementation of the analysis, shows and discusses benchmark results, and defines the global garbage collection. Chapter 7 summarizes the contributions of this dissertation, reviews related work, and gives an outlook on future work.

Appendix A explains the mathematical notation used in this dissertation. Appendix B contains the proof for the semantic correctness of the analysis. Finally, Appendices C and D list the transition rules for all PreScheme and Scheme primitive operations.

Chapter 2

Intermediate Language

The analyses developed in the following chapters operate on the intermediate language of Richard Kelsey's transformational compiler [*Kelsey*, 1989]. This compiler performs source-to-source transformations on an intermediate language that is based on the lambda calculus. The transformational compiler removes dependencies of the intermediate language semantics that the target machine cannot implement directly. For his thesis, Kelsey wrote Pascal and BASIC front-ends and gradually transformed the intermediate representation into a form that could easily be translated into MC68020 assembly code.

The heart of this compiler is its intermediate language and the transformation rules. In today's Scheme 48 system this compiler plays a significant role and hence is actively maintained. The transformational compiler is part of the *PreScheme* compiler [*Kelsey*, 1997] that translates PreScheme code, a statically typed subset of the Scheme programming language, to C code. The Scheme 48 virtual machine is written in PreScheme — this makes the transformational compiler indispensable for this system. Additionally, the Scheme 48 byte-code optimizer relies heavily on the transformational compiler's code simplifications.

Given these facts, choosing the transformational compiler's intermediate language as the input language for the analysis seems advantageous: One analysis may be used to analyze both, PreScheme code and Scheme code.

The specification of the intermediate language exists as comments in the compiler's source code. Additionally, a paper on the Scheme 48 native-code compiler [*Gasbichler and Sperber*, 2007] gives a brief introduction of the intermediate language for the purpose of describing the byte-code optimizer. This chapter introduces the syntax and semantics of the intermediate language and contributes a formal semantics that establishes a basis for developing a flow analysis.

2.1 Syntax

The intermediate language is a call-by-value lambda calculus in continuation-passing style (CPS) with the addition of constants and primitive operations. Thus, the language consists only of a few syntactic forms: lambda expressions, applications, variable references, and constants. Figure 2.1 summarizes the syntax of the intermediate language expressions.

$$
\begin{array}{llcl}
\mathit{lit} & \in \texttt{Lit}_{\mathit{Lang}} & = & \text{literals of the source language} \\
\tau & \in \texttt{Lab} & = & \text{set of labels} \\
v & \in \texttt{Var} & = & v^u \mid v^c \\
v^u & \in \texttt{Var}_u & = & \text{set of identifiers for user-defined variables} \\
v^c & \in \texttt{Var}_c & = & \text{set of identifiers for continuation variables} \\
g & \in \texttt{GVar} & = & \text{set of identifiers for global variables} \\
 & & & \text{with } \texttt{Var}_u \cap \texttt{Var}_c \cap \texttt{GVar} = \varnothing
\end{array}
$$

$$
\begin{array}{llcl}
\mathit{lam} & \in \texttt{Lam} & = & \mathit{lam}^u \mid \mathit{lam}^c \\
\mathit{lam}^u & \in \texttt{Lam}_u & = & (\lambda^{\mathsf{P}}\,(v_1^u \ldots v_n^u)\ \mathit{call})_\tau \\
\mathit{lam}^c & \in \texttt{Lam}_c & = & (\lambda^{\mathsf{C}}\,(v_1^c \ldots v_n^c)\ \mathit{call})_\tau \mid (\lambda^{\mathsf{J}}\,(v_1^c \ldots v_n^c)\ \mathit{call})_\tau \\
e & \in \texttt{Exp} & = & e^u \mid e^c \\
e^u & \in \texttt{Exp}_u & = & g \mid v^u \mid \mathit{lam}^u \mid \mathit{lit} \mid \mathit{tcall} \\
e^c & \in \texttt{Exp}_c & = & v^c \mid \mathit{lam}^c
\end{array}
$$

$$
\begin{array}{llcl}
\mathit{prim} & \in \texttt{Prim} & = & \texttt{Prim}_{\mathit{Lang}} \cup \texttt{Prim}_{\mathit{PCall}} \cup \texttt{Prim}_{\mathit{CCall}} \\
\mathit{prim}_p & \in \texttt{Prim}_{\mathit{PCall}} & = & \{\ \texttt{call}, \texttt{tail-call}, \texttt{unknown-call}, \\
 & & & \quad \texttt{unknown-tail-call}, \texttt{letrec1}, \texttt{letrec2}\ \} \\
\mathit{prim}_c & \in \texttt{Prim}_{\mathit{CCall}} & = & \{\texttt{return}, \texttt{unknown-return}, \texttt{jump}, \texttt{let}, \texttt{test}\} \\
\mathit{prim}_g & \in \texttt{Prim}_{\mathit{Glob}} & = & \{\texttt{global-ref}, \texttt{global-set!}\} \\
\mathit{prim}_l & \in \texttt{Prim}_{\mathit{Lang}} & = & \text{set of identifiers} \\
\mathit{tcall} & \in \texttt{TrivCall} & = & (\mathit{prim}\ \langle e_1^u, \ldots, e_n^u \rangle)_\tau
\end{array}
$$

$$
\begin{array}{llcl}
\mathit{call} & \in \texttt{Call} & = & \mathit{call}^u \mid \mathit{call}^c \\
\mathit{call}^u & \in \texttt{Call}_u & = & \left\{ \begin{array}{l}
(\mathit{prim}_u\ \langle c, f, a_1, \ldots, a_n \rangle)_\tau \\
\qquad \text{with} \quad c \in \texttt{Exp}_c\ f \in \texttt{Exp}_u\ a_i \in \texttt{Exp} \\
\qquad\qquad \mathit{prim}_u \in \left\{ \begin{array}{l} \texttt{call, tail-call,} \\ \texttt{unknown-call,} \\ \texttt{unknown-tail-call} \end{array} \right\} \\
(\texttt{letrec1}\ (\lambda^{\mathsf{C}}\ (v_1^c \ldots v_n^c) \\
\quad (\texttt{letrec2}\ \langle l^c, e_1^u, \ldots, e_n^u \rangle)_\tau)_{\tau'})_{\tau''}
\end{array} \right. \\
\mathit{call}^c & \in \texttt{Call}_c & = & \left\{ \begin{array}{l}
(\mathit{prim}_c\ \langle c, a_1, \ldots, a_n \rangle)_\tau \quad \text{with } c, a_i \in \texttt{Exp}_c \\
(\texttt{test}\ \langle c_0, c_1, a \rangle)_\tau
\end{array} \right.
\end{array}
$$

Figure 2.1: Intermediate language

Section 2.4 extends the intermediate language to facilitate the representation of complete PreScheme or Scheme programs. For the time being, however, a program is just a sequence of definitions. Each definition consists of a lambda expression or a literal expression and a global variable that names the value of

the expression. The language distinguishes three sorts of variables syntactically. Variables from $\mathtt{Var}_u$ (variables in the source program) and $\mathtt{Var}_c$ (variable introduced during the CPS transformation) are lexically bound variables introduced by lambda expressions. Global variables `GVar` reference the value of a definition. That is, the intermediate language syntactically distinguishes between variables visible only inside a lexical scope and variables that are globally visible.

Lambda expressions include an annotation that divides these expressions into three classes, all having the same semantics. The annotation describes how this lambda expression is used in a program: as a continuation, a user-defined procedure, or a jump target. A user-defined procedure is the translation of a procedure defined in the source program written by the programmer. The CPS conversion introduces continuation lambda terms and uses them as the continuation arguments for calls. A jump lambda is a continuation lambda whose call sites have been identified by the compiler and it is known that all call sites are within the same procedure. Jump lambdas are always arguments to `let` or `letrec` calls and either return via a continuation from the procedure they are part of or jump to another jump lambda within the procedure. These properties make jump lambdas particularly easy to implement: Jump lambdas become labels in the target code.

Classifying the lambda nodes by their origin as described above leads to a *partitioned CPS*. This differentiation is especially useful for implementing flow analyses that reason about the environment structure of programs [*Might and Shivers*, 2007]. The formal semantics developed in this chapter take these annotations into account to establish a basis for such analyses. However, the annotations are not essential for the formal semantics itself.

A lambda expression consists of a parameter list and a body. The body always consists of exactly one call. This call is a *non-trivial call*. Non-trivial calls only occur as the body of lambda expressions and are the most general form of an application. These calls consist of a *primitive operation* (*primop* for short) which is a constant that describes the nature of the call and an argument vector. Figure 2.1 lists all call primops.

The intermediate language includes a basic set of primops that implement and distinguish various kinds of procedure or continuation applications. These primops are *built-in primops* and establish the basic functionality of the intermediate language. However, for an input language like Scheme or PreScheme more primops are necessary. For example, a typical set of primops includes numerical and input/output operations. Hence, each front-end to the transformational compiler defines a set of *language-specific primops* and each back-end provides implementations of those primops in the target language. For the purpose of this chapter, language-specific primops do not matter and only built-in primops are of interest. Generally, calls have this form:

$$(\mathit{prim}\ \langle e^c_1, \ldots, e^c_n, e_1, \ldots, e_m \rangle)_\tau$$

That is, the argument vector consists of expressions (angle brackets denota a vector) that evaluate to continuation arguments e^c_i, followed by a list of arguments e_i that may evaluate to either continuation or user-world values. The number of continuations for a call depends on the primop being used: An ordinary procedure call always has one continuation argument — the continuation for the call. A conditional primop, however, splits the control flow: The `test`

primop, which corresponds to a Scheme `if` construct in the original program, has two continuation arguments; one for each branch.

Each lambda expression, call, and variable is annotated with a label τ. Each label is unique for a program and hence identifies a lambda expression, a certain call, or an occurence of a variable unambiguously. To keep the presentation clear, I willomit the labels attached to an expression when they are not relevant. Additionally, I use τ also in a notation that resembles a function application to improve clarity: $\tau(v^u)$ denotes the label τ of the variable v^u.

A *call* may either invoke a continuation, a jump lambda, or a user-defined procedure. The call may either take place in a tail context or not. The primop encodes both facts. Here is a list of all call primops:

- `unknown-call` This primop indicates that the compiler could not identify the lambda expression being called. A call with this primop follows this pattern:

  $$(\texttt{unknown-call}\ \langle l^c, e^u_1, \ldots, e^u_m \rangle)_\tau$$

  The compiler only knows two things about such calls: First, this call has a direct counterpart in the source program — it is a call to a user-defined procedure. Hence, the operator expression e^u_1 evaluates to a user-defined procedure. Second, in the source program the call occured in the context of another call which the CPS conversion turns into the continuation lambda l^c. Thus, the call is not a tail call.

- `call` Calls with this primop have the same form and properties as calls with `unknown-call`. However, the expression in operator position e^u is a lambda expression.

- `unknown-tail-call` This primop denounces a call to an unidentified user-defined lambda expression. These calls have this form:

  $$(\texttt{unknown-tail-call}\ \langle v^c, e^u_1, \ldots, e^u_m \rangle)_\tau$$

  The procedure being called is given by the expression e^u_1. In the source program this call appears at a tail position. Therefore, the call is a tail call and its continuation is the variable v^c.

- `tail-call` This primop is closely related to `unknown-tail-call`, but is used for tail-calls if the compiler could determine to which lambda expression the operator expression e^u evaluates.

- `unknown-return` This primop indicates a return via an unidentified continuation lambda given as e^c.

- `return` A call to a continuation lambda is usually called a `return` via a continuation. The form is:

  $$(\texttt{return}\ \langle e^c, e_1, \ldots, e_m \rangle)_\tau$$

  The `return` primop indicates a call to an operator e^c that is a continuation lambda (but not a jump lambda). Calls to continuations, like any other call, may have an arbitrary number of arguments. For `return` calls, the continuation lambda being called is known.

- `jump` This primop indicates a call to a jump lambda — that is, a lambda within the same procedure. The CPS representation distinguishes these calls from others because `jump` calls are particularly easy and fast to implement.[1]

- `let` Calls that use the `let` primop have this form:

  $$\texttt{(let } \langle l^c, e_1^u, \ldots, e_m^u \rangle\texttt{)}_\tau$$

  The `let` primop applies l^c to the values of the expressions e_1^u through e_n^u. Hence, it binds these expressions to variables. This primop exists to fulfill the requirement that every call has a primop.

- `test` The `test` primop implements the conditional execution of a continuation. A `test` call always has three arguments: two continuation lambda expressionss with zero arity (l_0^c and l_1^c) and a test expression e^u:

  $$\texttt{(test } \langle l_0^c, l_1^c, e^u \rangle\texttt{)}_\tau$$

To implement local recursive binding, the intermediate language uses `letrec`. The representation of `letrec` is a combination of two nested calls with primops `letrec1` and `letrec2`:

$$\texttt{(letrec1 } \langle l^c \texttt{ (}\lambda^{\texttt{C}}\texttt{ (}v_1^u \ldots v_n^u\texttt{)} \;\; \texttt{(letrec2 } \langle l_{body}^c\; l_1^u \ldots l_n^u \rangle\texttt{)}_\tau\texttt{)}_{\tau'} \rangle\texttt{)}_{\tau''}$$

The variables to bind (v_1^u through v_n^u) are represented as arguments to the continuation of the `letrec1` call. The body of the continuation contains a call to `letrec2` and represents the body of `letrec` in its continuation l_{body}^c and the values to bind as the arguments l_1^u through l_n^u. In terms of ordinary `letrec` expressions the variables v_i^u are the left-hand sides of the bindings, the expressions l_i^u the right-hand sides of the bindings, and l_{body}^c relates to the body of the `letrec`.

Besides the calls just discussed, the intermediate language has *trivial calls*: These calls do not have a continuation argument and are together with variable references and lambda expressions a further type of expressions. Non-trivial calls may have trivial calls as their arguments, and the arguments of a trivial call may contain further trivial call arguments. Trivial calls simply compute values, have no side effects, and do not affect the control flow. Most trivial calls employ a primop that is specific to the input language. The following example is the intermediate representation of an excerpt from a PreScheme program:[2]

```
(test c_13 c_14 (address< a_15 (global-ref *from-end*)))
```

The conditional call `test` either calls the continuation `c_13` or `c_14`. The test expression is a trivial call to the PreScheme specific address-comparing primop `address<`. The second argument to `address<` is also a trivial call to `global-ref`, the primop that accesses a global variable.

[1] The C back end of PreScheme compiler implements `jump` calls using C's `goto` construct.
[2] The excerpt is taken from the source code of Scheme 48's twospace garbage collector.

2.2 Printing CPS Programs

The following chapters present examples of programs written in the intermediate language rather than the source language. CPS programs have a deeply nested structure which makes them hard to read: For example, most calls have lambda expressions as their continuation arguments, which themselves contain calls. The transformational compiler comes with a pretty-printer that outputs the abstract syntax tree in a more readable form. The source code presented in course of this work always follows this representation.

Here is a very simple program written in the intermediate language:

```
(lambda (k x)
  (test (lambda ()
          (unknown-return k 42))
        (lambda ()
          (unknown-return k 23))
        x))
```

In this example there is one call that uses the `test` primop with three arguments: two continuation lambda expressions and a variable reference to `x`. Both continuation nodes return via the continuation provided as the argument `k` and return constants. The pretty-printer displays this program as follows:

```
0   (P p_0 (k x)
       (test 2 ^c_1 ^c_2 x))

1      (C c_1 ()
          (unknown-return 0 k '42))

2      (C c_2 ()
          (unknown-return 0 k '23))
```

The first thing to note is that both continuation lambda expressions no longer appear as arguments to the `test` call. The pretty-printer has inserted a *label* instead: `^c_1` stands for the continuation lambda node with the identification number 1. The leading circumflex character `^` distinguishes labels from variable references.

The indention of the program is important. The continuation nodes have the same level of indention as the body of the outer lambda expression: That is, the continuations are in the scope of the outer lambda expression.

When printing lambda expressions the pretty-printer omits the lambda symbol but prints the type of the term instead: `P` stands for user-defined procedures denoted as $\lambda^{\mathtt{P}}$ in the syntax description of Figure 2.1, `C` for continuation lambdas, and `J` for jump lambda. The compiler assigns a unique number to all lambda expressions to identify them easily (`p_0` in this case). The pretty-printer prints this number to the left of each lambda expression. Additionally, user-defined procedures may have a name. The name helps the user to identify the CPS lambda that corresponds to the original procedure in the direct-style source code. Hence, $(\lambda^{\mathtt{P}}_{name}\ (args)\ call)$ prints as follows:

```
id      (P name_id (args)
           call)
```

The output for applications is straightforward. An application yields the following output:

(*prim* *n* a_1 ... a_m)

Each application includes the primop being used as the first form inside the parentheses followed by a number. This number indicates the number of continuation arguments. For the `test` primop, this number is always two, for procedure calls one, and for continuation returns zero.

The following PreScheme code computes the factorial of a number using the tail-recursive loop `lp`:[3]

```
(define (fact n)
  (let lp ((n n) (r 1))
    (if (< n 2)
        r
        (lp (- n 1) (* n r)))))

(define (main)
  (fact 10))
```

The pretty-printer then prints the program produced by the PreScheme front end as follows:

```
22  (P main_22 (c_21)
28    (LET* (((x_26 lp_27)
                          (letrec1))
34           (()          (letrec2 x_26 ^lp_31)))
        (jump 0 lp_27 '10 '1)))

31   (J lp_31 (n_29 r_30)
       (test 2 ^c_33 ^c_32 (< n_29 '2)))

33    (C c_33 ()
        (unknown-return 0 c_21 r_30))

32    (C c_32 ()
        (jump 0 lp_27 (+ '-1 n_29) (* n_29 r_30)))
```

The pretty-printer displays calls to `letrec1` and `letrec2` in a special condensed form that needs some explanation. As discussed in the preceding section, the representation for local recursive bindings involves two nested calls. The compiler adds an unique identifying variable —`x_26` in this case — to both calls. If some transformation in the further course of the compiler run accidently separates these calls, this mistake may be discovered by comparing the identifying variables of both calls. The pretty-printer displays `letrec` in form of a `LET*`. Continuation lambda `c_34` is the body of the `letrec` expression. The pretty-printer prints the variables to bind in form of the tuple `(x_26 lp_27)` on the left side of a `LET*` binding pair. An additional asterisk `*` after the names of the bound variables indicates a call that is not `let`.

[3] The loop is expressed using *named let* (a standad Scheme macro) which combines a `letrec` defined procedure with a call to that procedure.

$$
\begin{array}{llcl}
\varsigma & \in \mathbf{State} & = & \mathbf{Eval} + \mathbf{Apply} \\
& \mathbf{Eval} & = & \mathtt{Call} \times \mathbf{BEnv} \times \mathbf{VEnv} \times \mathbf{Store} \times \mathbf{Time} \\
& \mathbf{Apply} & = & \mathbf{Proc} \times \mathbf{D}^* \times \mathbf{D}^* \times \mathbf{VEnv} \times \mathbf{Store} \times \mathbf{Time} \\
\beta & \in \mathbf{BEnv} & = & \mathtt{Var} \rightarrow \mathbf{Time} \\
ve & \in \mathbf{VEnv} & = & \mathtt{Var} \times \mathbf{Time} \rightarrow \mathbf{D} \\
& \mathbf{HLoc} & = & \mathtt{Lab} \times \mathbf{Time} \\
& \mathbf{GLoc} & = & \mathtt{Var} \\
loc & \in \mathbf{Loc} & = & \mathbf{HLoc} + \mathbf{GLoc} \\
ref & \in \mathbf{Ref} & = & \mathbf{Loc} \times \mathtt{Lit}_{Lang} \\
\sigma & \in \mathbf{Store} & = & \mathbf{Ref} \rightarrow \mathbf{D} \\
proc & \in \mathbf{Proc} & = & \mathbf{Clo} + \{\mathtt{halt}\} \\
clos & \in \mathbf{Clo} & = & \mathtt{Lam} \times \mathbf{BEnv} \times \mathbf{Time} \\
& \mathbf{D} & = & \mathbf{Proc} + \mathbf{Ref} + \mathbf{Bas}_{Lang} + \mathbf{Comp}_{Lang} \\
t & \in \mathbf{Time} & = & \mathbb{N}
\end{array}
$$

Figure 2.2: CPS language semantic domains

2.3 Semantics

The classic approach for systematically designing a control flow analysis involves abstracting the formal semantics of a programming language. The semantics defined in this section serve as a basis for a systematic design of the control-flow analysis (see Chapter 4).

The semantics is a small-step operational semantics which is similar to the semantics Might and Shivers [*Might and Shivers*, 2007, 2006] define in preparation for their environment analysis. Therefore, many of their techniques for flow analyses such as abstract garbage collection or environment analysis with frame strings also work with this analysis (see Chapter 6). In fact, Might and Shivers consider a partitioned CPS very similar to the transformational compiler's intermediate language. Hence, expressing the semantics in a similar way is a natural choice.

Figure 2.2 shows the semantic domains. The machine that executes the program is either in an *eval* or in an *apply* state. The mode of operation is simple: During an *apply* state the machine binds the parameters of the procedures to the argument values and produces an *eval* state to evaluate the body (a call) of the procedure. Hence, the machine executes the program as an alternating sequence of *eval* and *apply* states. During an *eval* state the machine evaluates the operator expression and all arguments to the call and stores these values in an *apply* state tuple.

However, before considering the transition rules, consider the semantic domains in Figure 2.2. The values in **D** are the denotable values. The exact structure of this set is only of minor importance to the transformations carried out by the transformational compiler. The compiler simplifies the structure of the program but never involves the values of a program directly. Values and types become important in later phases of the compilation and the analysis, however. For the moment, it is sufficient to assume that the denotable values consist of function values **Proc**, references **Ref** to values stored in a heap, basic values $\mathbf{Bas}_{Lang}$ (e. g. integers and characters) and compound values $\mathbf{Comp}_{Lang}$

(e. g. records and vectors) that combine multiple denotable values. Section 4.4 defines the language specific semantic domains for Scheme and PreScheme.

A procedure value **Proc** is either a closure **Clo** or the halt continuation. The `halt` continuation is a special procedure value that, if applied, stops the program. A closure has three components: The lambda expression that produces the closure, a value from **Time** that denotes the point in time when the closure was created, and a binding environment. For the standard semantics defined in this section, the set **Time** is an ordered denumerable set of values. Furthermore, **Time** includes a start time t_0 and the function

$$\mathit{tick} : \mathbf{Time} \rightarrow \mathbf{Time}$$

that advances time by one step.

Consider the second constituent of a closure triple: The binding environment. A binding environment maps variables to points in time — the time a binding was established. The value bound to the variable, however, is not stored in a binding environment but in a *variable environment* **VEnv**. A variable environment maps a variable and a binding time to value. Hence, looking up a variable is a two-step process: First, the binding environment maps the variable name to a binding time. The second step involves the variable environment: Looking up the variable together with the binding time yields the value of the variable. Hence, a variable lookup in the semantics always has this form:

$$ve(v, \beta(v)) \in \mathbf{D}$$

Splitting the environment into a binding environment and variable environment avoids the troubles that would arise from a recursive domain for the environment [*Shivers*, 1991]. (The straightforward — and recursive — definition would be $\mathbf{Env} = \mathtt{Var} \rightarrow \mathbf{D}$ where the summand for closures in **D** would contain an environment: $\mathbf{Clo} = \mathtt{Lam} \times \mathbf{Env}$.)

Besides lexically bound variables, the intermediate language supports global variables and values stored in a heap. In contrast to lexically bound variables, global variables and heap values are mutable. In the semantics, the domain **Store** models the heap. The store is a mapping from references to values. References **Ref** are comparable to pointers in a programming language and identify a value in the heap. A reference consists of a location and a so-called *selector* (which is always a literal). This partitioning makes representing compound values such as vectors or records that reside in the heap easy: The location identifies the compound value as a whole and a reference identifies a single value slot within a compound value. That is, the location may be regarded as a base pointer and the literal value as an offset.

A location either is the location of some heap value (**HLoc**) or the location of a value bound to a global variable (**GLoc**). The locations of heap values are identified by the label of the call, which allocates the heap space, and the point in time the call occurs.

Distinguishing references, heap locations, and global locations may seem overly complicated at this point, however, for the purpose of the analysis maintaining this extra complexity is worthwhile. As discussed in Chapter 4, the flow analysis collapses multiple distinct states in the concrete semantics into a single abstract state. That is, a location in the abstract semantics represents multiple locations in the concrete semantics and abstraction functions assign concrete to

$$
\begin{array}{lcl}
\mathcal{A}\,\beta\;ve\;\sigma\;t\;lam & = & (lam, \beta, t) \\
\mathcal{A}\,\beta\;ve\;\sigma\;t\;v & = & ve(v, \beta(v)) \\
\mathcal{A}\,\beta\;ve\;\sigma\;t\;g & = & \sigma(g, \top) \\
\mathcal{A}\,\beta\;ve\;\sigma\;t\;lit & = & \mathcal{K}_{Lang}\;lit \\
\\
\mathcal{A}\,\beta\;ve\;\sigma\;t\;(prim\;\langle e^u_1, \ldots, e^u_n\rangle) & = & \mathcal{P}_{Lang}\;\beta\;ve\;\sigma\;t\;prim\;\langle e^u_1, \ldots, e^u_n\rangle
\end{array}
$$

Figure 2.3: Evaluating call arguments

abstract locations. To control the precision of the analysis the user may wish to run the analysis with different abstractions for locations. Hence, it is advantageous to separate references and locations to make exchanging the definition of locations easier.

Consider the definition of **State** again. *Apply* states reflect the situation just before binding the arguments of a procedure. Hence, these states include a procedure value and the arguments to bind. Note that *apply* states contain two argument vectors $\mathbf{D}^*$: The first vector only contains continuation arguments, the second contains "user-world arguments". Recall that the conversion to CPS adds an additional argument to each call: the continuation of the call (see Figure 2.2). That is, there are call arguments that transport continuations, so called *continuation arguments*, and call arguments that transport user-world values, the *user-world arguments*. Keeping these values in separate vectors in *apply* states is not vital for the semantics. It is, however, useful for the analysis: The implementation of the frame string analysis, for example, often needs to distinguish values by this category. Hence, keeping them separate in the first place speeds up the analysis and makes the implementation easier to understand.

The transition rules described later in this section thread the variable environment **VEnv** from *eval* to *apply* states. Thus, the variable environment is a component of both state types. Threading the binding environments through the semantics, in contrast, follows this scheme: Closures include a binding environment, and the transition rules ensure that this binding environment will be used to evaluate the procedure body. Basically, this means the binding environment from the closure in an *apply* state flows into an *eval* state. Hence, there is no need for an *apply* state to have a separate **BEnv** component.

Figure 2.3 shows the semantic function $\mathcal{A}$ that evaluates the arguments of a call. Call arguments are either lambda expressions, variable references, literals, or trivial calls. To evaluate a lambda expression or a variable reference $\mathcal{A}$ constructs a closure or finds the value in a given binding environment and variable environment, respectively. If the variable is a global variable v^g, $\mathcal{A}$ accesses the store. As the intermediate language does not know about the basic values of the source language it shifts the burden of evaluating *lit* to the language specific evaluation function $\mathcal{K}_{Lang}$.

Trivial calls appear as arguments of regular calls, or as arguments of other trivial calls. Therefore, the evaluation function $\mathcal{A}$ needs to take them into account. $\mathcal{A}$ delegates the work to $\mathcal{P}_{Lang}$ that is an interface to the language-dependent part of the semantics: Each input language defines $\mathcal{P}_{Lang}$ for its primops. The primops of $\texttt{Prim}_{CCall}$ and $\texttt{Prim}_{PCall}$ alter the control flow, and

$$\frac{prim_p \in \texttt{Prim}_{PCall} \setminus \{\texttt{letrec1}, \texttt{letrec2}\}}{(\texttt{(}prim_p\ \langle c, f, a_1, \ldots, a_n\rangle\texttt{)}_\tau, \beta, ve, \sigma, t) \rightharpoonup (proc, \langle c' \rangle, d^*, ve, \sigma, t')} \quad \text{(PCallEval)}$$

$$\text{where} \begin{cases} t' &= tick(t) \\ proc &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ f \\ c' &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ c \\ d_i &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ a_i \end{cases}$$

$$\frac{prim_c \in \texttt{Prim}_{CCall} \setminus \{\texttt{test}\}}{(\texttt{(}prim_c\ \langle c, a_1, \ldots, a_n\rangle\texttt{)}_\tau, \beta, ve, \sigma, t) \rightharpoonup (proc, \langle\rangle, d^*, ve, \sigma, t')} \quad \text{(CCallEval)}$$

$$\text{where} \begin{cases} t' &= tick(t) \\ proc &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ c \\ d_i &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ a_i \end{cases}$$

$$\frac{}{(\texttt{(}prim_l\ \langle c, a_1, \ldots, a_n\rangle\texttt{)}_\tau, \beta, ve, \sigma, t) \rightharpoonup (c', \langle\rangle, \langle r\rangle, ve, \sigma', t')} \quad \text{(PrimCallEval)}$$

$$\text{where} \begin{cases} t' &= tick(t) \\ c' &= \mathcal{A}\,\beta\ ve\ \sigma\ t'\ c \\ (r, \sigma') &= \mathcal{P}_{Lang}\,\beta\ ve\ \sigma\ t'\ prim_l\ \langle a_0, \ldots, a_n\rangle \end{cases}$$

Figure 2.4: Simple transitions for *eval* states.

hence may not be used in the context of a trivial call. Note that $\mathcal{A}$ does not evaluate the arguments of the trivial call; this burden is also shifted to $\mathcal{P}_{Lang}$. Most primops evaluate their arguments using $\mathcal{A}$. However, some primops treat their arguments specially. `Global-ref` is an example: This (Scheme and PreScheme) primop expects a variable as its argument and accesses the global environment. Hence, `global-ref` needs the unevaluated argument. Evaluating the arguments with $\mathcal{A}$ before passing them to $\mathcal{P}_{Lang}$ would make it impossible to define such a primop as a case for $\mathcal{P}_{Lang}$.

Now the necessary machinery for writing down the state transition rules for *eval* states is in place. Figure 2.4 shows the transition rules for *eval* states.

All three rules evaluate calls. If the primop indicates that the call moves the control flow to a user-defined procedure then PCallEval applies. (Except if the primop is `letrec1` or `letrec2` — these primops have a separate rule.) PCallEval decomposes the argument vector into three components: The procedure to call f, the continuation c of the call, and the remaining arguments of the call. In a second step PCallEval evaluates all these ingredients using the argument evaluation function $\mathcal{A}$. Finally, PCallEval constructs a new *apply* state using the evaluated operator expression *proc*, the evaluated continuation expression c', and the evaluated vector of remaining arguments d^*. The rule CCallEval works similarly. However, this rule applies to calls that use a $\texttt{Prim}_{CCall}$ primop (except `test`) and subsequently decomposes the argument vector differently.

$$\frac{}{(\texttt{(test}\ \langle c_0, c_1, a\rangle)_\tau, \beta, ve, \sigma, t) \rightharpoonup (c', \langle\rangle, \langle\rangle, ve, \sigma, t')} \quad \text{(TestEval)}$$

$$\text{where} \begin{cases} t' &= tick(t) \\ r &= \mathcal{A}\ \beta\ ve\ \sigma\ t'\ a \\ c' &= \begin{cases} \mathcal{A}\ \beta\ ve\ \sigma\ t'\ c_0 & \text{iff}\ \ trueish?(r) \\ \mathcal{A}\ \beta\ ve\ \sigma\ t'\ c_1 & \text{otherwise} \end{cases} \end{cases}$$

$$\frac{}{\begin{array}{l}(\texttt{(letrec1}\ \langle(\lambda\ (v_1^c \ldots v_n^c) \\ \quad \texttt{(letrec2}\ \langle l^c, e_1^u \ldots e_n^u\rangle)_\tau\rangle)_{\tau'})_{\tau''}, \beta, ve, \sigma, t)\end{array} \rightharpoonup (call, \beta', ve', \sigma, t')} \quad \text{(LetrecEval)}$$

$$\text{where} \begin{cases} (\lambda^{\mathsf{c}}()\ \ call) &= l_c \\ r_i &= \mathcal{A}\ \beta'\ ve\ \sigma\ t'\ e_i^u \\ \beta' &= \beta[v_i \mapsto t'] \\ ve' &= ve[(v_i^c, t') \mapsto r_i] \\ t' &= tick(t) \end{cases}$$

Figure 2.5: Complex transitions for *eval* states

The primop of the call may be specific to the source language — the rule PrimCallEval reflects this situation. This rule shifts the burden of evaluating the call completely to the function $\mathcal{P}_{Lang}$ which, as just shown, is also responsible for evaluating calls with language specific primops. Note, that in this context $\mathcal{P}_{Lang}$ returns a tuple consisting of a value r and a store σ' — this allows the definition of language specific primops that modify the store. Finally, PrimCallEval calls the continuation c' with the value r computed by $\mathcal{P}_{Lang}$.

Figure 2.5 shows the transitions for calls using the `letrec` and `test` primops. The conditional `test` primop is the only compiler specific primop that requires knowledge about the data types of the source language. First, the rule evaluates the test expression a and based on the value of this expression the rule calls the consequent continuation lambda expression c_0 or the alternative lambda expression c_1. To decide which continuation to call, `test` employs a function named *trueish?*. The definition of *trueish?* is specific to the source language and recognizes the values that are considered to be true in the source language semantics. The Scheme language, for example, considers all values except `#f` to be true, hence the definition for *trueish?* would just compare its argument value with `#f`.

In contrast to the transition rules seen so far, the LetrecEval rule of Figure 2.5 transfers the machine from an *eval* state to an *eval* state. This is particularly useful in this case, because this rule sets up the environments with the bindings of the `letrec` call and then continues the evaluation with an *eval* state for the call in the body of `letrec`. Recall the special representation of `letrec` in the transformational compiler's intermediate language from Section 2.1: It uses two nested calls to the `letrec1` and `letrec2` primops. The parameters of the continuation of the outer call are the left-hand sides of the `letrec` bindings and the call arguments of the inner call are the right-hand sides. The body of

$$\frac{}{((\texttt{global-ref}\ \langle c, g\rangle)_\tau, \beta, ve, \sigma, t) \twoheadrightarrow (proc, \langle r\rangle, \langle\rangle, ve, \sigma, t')} \quad \text{(PGlobalRef)}$$

$$\text{where} \begin{cases} t' & = tick(t) \\ proc & = \mathcal{A}\,\beta\ ve\ \sigma\ t'\ c \\ r & = \sigma(g, \top) \end{cases}$$

$$\frac{}{((\texttt{global-set!}\ \langle c, g, a\rangle)_\tau, \beta, ve, \sigma, t) \twoheadrightarrow (proc, \langle\rangle, \langle\rangle, ve, \sigma', t')} \quad \text{(PGlobalSet)}$$

$$\text{where} \begin{cases} t' & = tick(t) \\ proc & = \mathcal{A}\,\beta\ ve\ \sigma\ t'\ c \\ d & = \mathcal{A}\,\beta\ ve\ \sigma\ t'\ a \\ \sigma' & = \sigma[(g, \top) \mapsto d] \end{cases}$$

Figure 2.6: Transitions for accessing the store

`letrec` is encoded as the continuation of the inner call, which is l^c in this case. Hence, the call in the resulting *eval* state is the call from in the body of l^c. The call takes place under the augmented environments β' and ve'. These environments include the new bindings. The introduction of this section claimed that *eval* states evaluate call arguments, and *apply* states bind the results. The LetrecEval rule, however, is more complex and is the result of a fusion of two consecutive state transitions.

Figure 2.6 shows the transition rules for calls to the primops `global-set!` and `global-ref`. Consider PGlobalSet first. This rule updates the values for the variable v^g with the value of a in store σ. This leads to a new store σ'. Then the rule constructs an *apply* state for each continuation with the new store. Since PGlobalSet is a rule that starts from an *eval* state, it also advances the time.

Accessing a global variable is slightly more complex since this happens in three different contexts: A regular call using `global-ref`, a trivial call using `global-ref`, or call argument that is a reference to a global variable. The transition rule PGlobalRef depicts the first situation. Here, the primop `global-ref` occurs as the primop to a regular call. The rule proceeds in three steps: advance the time, find the value of the global variable in the store, and call all continuations with the value of global variable as the only argument.

The second situation concerns argument evaluation. Call arguments are expressions and consequently a global variable may also be used as an argument. That is, the argument evaluation function $\mathcal{A}$ must also be prepared to access the store (see Figure 2.3). Finally, `global-ref` may also appear in a trivial call. In this case, $\mathcal{P}_{Lang}$ is responsible for the evaluation. Hence, $\mathcal{P}_{Lang}$ must have a case that accesses the store:

$$\mathcal{P}_{Lang}\,\beta\ ve\ \sigma\ t\ \texttt{global-ref}\ g = \sigma(g, \top)$$

The ApplyClos rule of Figure 2.7 extends the variable and binding environment with the call arguments. This rule presumes that the number of parameters of the lambda expression matches the number of arguments given by the

$$\frac{\text{length}(c^*\S d^*) = n}{(((\lambda\ (p_1 \ldots p_n)\ \mathit{call})_\tau, \beta, t_b), c^*, d^*, ve, \sigma, t) \twoheadrightarrow (\mathit{call}, \beta', ve', \sigma, t)} \quad \text{(ApplyClos)}$$

$$\text{where} \begin{cases} \beta' & = \beta[p_i \mapsto t] \\ ve' & = ve[(p_i, t) \mapsto (c^*\S d^*)_i] \end{cases}$$

Figure 2.7: Transition for *apply* states

continuation argument vector c^* and the user argument vector d^*. (The operator § concatenates two vectors. See Appendix A for overview on notation.) In a first step the rule decomposes the closure into the binding environment β over which the lambda expression is closed, the procedure parameters p_i, and the procedure body *call*. The result of this transition is an *eval* state for *call* with the extended environments. Hence, the rule first updates (in the sense of functional updates) the binding environment β for each parameter p_i. The bound values reside in the new variable environment *ve*. Both environments β' and ve' together then reflect the situation in which *call* occurs.

2.4 Complete programs

The preceding sections defined the semantics of the CPS intermediate language. However, the semantics laid out in this chapter are merely building blocks — so far there is no notion for a complete program. The transformational compiler has no built-in notion of a complete program, but instead works on individual CPS nodes. The compiler shifts the responsibility for handling complete programs to the front end. Hence, different notions are possible for different source languages. This section reviews the notions of the PreScheme and Scheme front ends.

2.4.1 Complete PreScheme programs

To compile a PreScheme program the user provides a Scheme 48 module definition file and the name of a procedure to the compiler. The procedure marks the start of control. The code resides in source files tied together with a Scheme 48 module file: At compile time, the front end loads this module file and evaluates the expressions at top level. The front end then translates the top level definitions into CPS intermediate language for further treatment by the transformational compiler.

Each top level definition is a pair consisting of the definition's name and the value being bound — in terms of the front-end this is a *definition*. The front end stores some extra information such as the dependencies and the value type of this definition with each definition. This information, however, is not of interest for the flow analysis.

A complete description of the PreScheme semantics must include the semantics for evaluating the top-level expressions: the *compile-time semantics*. However, for designing a flow analysis, these semantics are of no interest. In a first step, the compiler evaluates the top-level expressions of the program —

that is, *before* the conversion to CPS intermediate representation. Consequently, the CPS conversion translates a program with evaluated top-level expressions to CPS rather than a source file. The flow analysis operates on the CPS representation of the program, and thus is independent of PreScheme's compile-time semantics.

2.4.2 Complete Scheme programs

Scheme programs are not given in the form of a source file read by a front end. As discussed in Section 1.3, the byte-code compiler compiles the source text to a byte-code version of the program and the byte-code optimizer subsequently translates byte code to CPS intermediate languages. This translation works on a procedure basis: Given the reference to a procedure, the optimizer's front end generates a CPS syntax tree for this procedure.

This approach is suitable for the context of the byte-code optimizer: The Scheme 48 compiler aims at translating the procedures of a program from byte code to native code. The system is designed such that procedures compiled to native code may call byte-code procedures and vice versa [*Gasbichler and Sperber*, 2007].

However, this causes some problems for a flow analysis: Other procedures referenced in this procedure need to be available to the flow analysis also in the form of CPS intermediate language. To overcome this problem, the analysis first traverses the byte code and builds the transitive closure over all free variables in the procedure. The result is a list where each entry consists of a so-called *location* and the code of the procedure converted to CPS. A location is a unique reference within Scheme 48.

Another challenge comes from the fact that these procedures usually contain references to values of arbitrary types: integers, strings, lists, records, and so on. Besides being able to abstract literal nodes in source code to abstract values, the abstraction functions of the analysis must also be capable of abstracting given concrete values to abstract values. Section 4.4 discusses these abstraction functions.

2.4.3 Treating complete programs

As the previous sections illustrate, programs consist of a set of globally accessible definitions and a start of control. To model a complete program two additional domains are necessary:

$$
\begin{array}{llcl}
f & \in \texttt{Definition} & = & \texttt{GVar} \times \texttt{Exp} \\
prg & \in \texttt{Prog} & = & \texttt{Definition}^* \times \texttt{Lam}
\end{array}
$$

A *form* models a top-level notation consisting of a variable name and an expression. Note that the expressions are the right-hand sides of a definition and are lambda expressions, literals, or variable references. The lambda node in a *prg* tuple specifies the start of control and must have no arguments.

This notion defines a common denominator for representing a complete program, leaving open one important semantic question: The dependencies among these forms. The transformational compiler is mainly concerned with the translation of procedures and hence does not establish a semantic discipline for forms.

However, a flow analysis always concerns a complete program. Thus, a semantic description of the interaction of forms is indispensable.

The flow analysis assumes that it is possible to rewrite the program such that all global bindings are bound using one `letrec*` [*Waddell et al.*, 2005], a variant of the Scheme `letrec` binding construct, that wraps the program:

```
(letrec* ((v1 e1)
          ...
          (vn en))
  (main))
```

`letrec*` allows defining mutually recursive procedures and values: The variable v_i introduced by the binding pair `(vi ei)` may be used on every right side of a binding pair of the same `letrec*` and its body. The variable `vi` is initialized with the value of the corresponding expression `ei`. However, there is one important restriction: It must be possible to evaluate the right-hand sides of the bindings in a left to right order without referring to the corresponding variable or any binding that follows in the binding list. [*Sperber et al.*, 2007]

Hence, the order of the definitions is important and the definitions of a program are kept in a vector rather than a set. The PreScheme front-end already sorts the definitions according to this restriction. When analyzing a Scheme program, the pre-pass that finds all dependent global definitions sorts these dependencies topologically. Thus, the pre-pass puts the definitions in the required order.

2.4.4 Initial state

The evaluation starts with an *initial state* computed by the injection function I. The initial state is an *apply* state that applies the main function to an empty argument vector with the `halt` continuation as continuation argument:

$$I : \texttt{Prog} \mapsto \textbf{State}$$
$$I(\langle f^*, l_{main}\rangle) = \langle\langle l_{main}, \beta_0, t_0\rangle, \langle\rangle, \langle\{\texttt{halt}\}\rangle, ve_0, \sigma_I, t_0\rangle$$

The environments β_0 and ve_0 are empty and t_0 is the first point in time. σ_I is the initial state that contains all global definitions. That is, the right-hand side of the forms of a program are not bound as lexical variables but using the store — because a program may mutate the global variables.

Setting up the initial store σ_I works by gradually extending an empty store as follows:

$$\begin{aligned} \sigma_0 &= [] \\ \sigma_i &= \sigma_{i-1}[(g_i, \top) \mapsto \mathcal{A}\, \beta_0\, ve_0\, \sigma_{i-1}\, e_i\,] \\ \sigma_I &= \sigma_n \\ & \quad \text{where} \quad \forall (g_i, e_i) = f_i^*, f^* \in \texttt{Definition}^* \end{aligned}$$

This equation system assumes that the definitions f^* with the global variables g_i are ordered ascending according to the `letrec*` property specified in the preceding section. Equation i evaluates the right-hand side of the definition i, the expression e_i, under the empty binding environment, the empty variable environment, and the preceding store $\widehat{\sigma}_{i-1}$ using $\widehat{\mathcal{A}}$. This scheme corresponds to a left-to-right evaluation of a `letrec*` form.

Chapter 3

Flow analysis of Programs

Flow analyses determine aspects of the run-time behavior and properties of programs at compile time. The results of a flow analysis are called *flow information*. Generally, flow analyses fall into two categories:

- *Control-flow analyses* determine how the control advances through the program at run time.
- *Data-flow analyses* determine which values a program may compute and how the program passes values from one part of the program to another.

For higher-order languages, control-flow analyses are intrinsically tied to data-flow analyses: Procedures are first-class values, and flow like other values through the program. Hence, knowing which procedure values reach which program points — a typical result of data-flow analysis — is a necessary prerequisite for determining the control flow.

This chapter gives a short and informal introduction to control-flow analysis for higher-order languages.

3.1 Control flow

A control-flow analysis computes the *flow graph* of a program. The flow graph characterizes in which order the basic blocks evaluate at run-time. The basic blocks of the intermediate language presented in Chapter 2 are procedures. Hence, in this case the flow graph shows which procedures call which other continuations or procedures.

The nodes of a flow graph correspond to the basic blocks. An edge encodes to which basic block the control may transfer during a program run. It is important to understand that the flow graph computed by a flow analysis does not illustrate the control transfers of a particular program run. Instead, the flow graph depicts the control transfers for *all possible program runs*. This property is very important as it establishes a foundation for the compiler to reason and subsequently to optimize a program for all possible runs.

Consider Figure 3.1 for an example: This figure shows a PreScheme program in its intermediate representation on the left-hand side and its flow graph on the right-hand side. The flow graph depicts continuation and jump lambdas as white

```
8  (P square_8 (c_6 x_7)
     (unknown-return 0 c_6 (* x_7 x_7)))

46 (P main_46 (c_45)
20   (LET* (((v_19)* (make-vector '10 '0))
43          ((x_42 lp_21)
                (letrec1))
41          (()   (letrec2 x_42 ^lp_25)))
       (jump 0 lp_21 '0 square_8)))

25  (J lp_25 (i_23 p_24)
      (test 2 ^c_28 ^c_29 (< '10 i_23)))

28   (C c_28 ()
       (unknown-return 0 c_45 v_19))

29   (C c_29 ()
31     (LET* (((v_30) (unknown-call p_24 i_23))
33            ((v_32) (vector-set! v_19 i_23 v_30)))
         (jump 0 lp_21 (+ '1 i_23) p_24)))
```

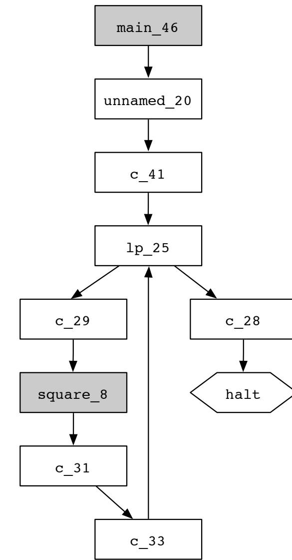

Figure 3.1: Example: flow graph

nodes, user-defined procedures as grey nodes, and the special halt continuation with a hexagon. Consider the program first. The program computes a vector of length ten that contains the first ten square numbers. There are two definitions involved: A procedure `square_8` that computes x^2 for an argument x, and the more complex procedure `main_46`.

The procedure `main_46` allocates a fresh vector and binds it to `v_19`. Next, the evaluation of `letrec` starts and binds a closure over the procedure `lp_25` to the variable `lp_21`. The body of the `letrec` is continuation lambda `c_41`, which starts the recursive computation by jumping to lambda `lp_25`.

Now the control reaches `lp_25`. The body of this jump lambda contains a call to the `test` primop to dispatch on the value of `i_23`. From this point the control — depending on the value of `i_23` — can follow two distinct paths: Either proceed to `c_28` and stop the recursive computation or continue to `c_29` for the next iteration. Since the flow graph, as stated before, shows all possible control transfers the according node has two outgoing edges — one for each continuation.

Basically, `lp_25` implements a loop that counts from 1 to 10 using the parameter `i_23`. The second parameter, `p_24` is the procedure that computes the number to store in the vector. A closer inspection of the loop reveals that the first call to `lp_25` passes `square_8` as the value for `p_24` and every subsequent call to `lp_25` just passes the value without altering. Hence, applying `p_24` can only mean applying `square_8`. However, the compiler fails to notice this simple fact: The call in the body of `c_31` is an unknown call.

The flow graph, however, reveals to which lambda form the control proceeds to after the evaluation of `c_29`: `square_8`. Since the flow graph shows the control transfers for all program runs it is safe to assume that at this call site `square_8` is the only target.

3.2 Using control-flow information for optimization

Compilers use flow information to drive and guide optimizations. This applies chiefly to *high-level optimizations* that take place in early phases of the compilation process. This section exemplifies this interplay by means of an example: *inlining*, sometimes also called *inline expansion*.

Flow information is not a prerequisite for all optimizing program transformations. Especially low-level optimizations that operate on machine code or an intermediate representation close to the final code often do not depend on flow information. Optimizations that reorder machine code to exploit parallelism, various loop transformations, or generating hints for the branch prediction mechanism are examples for such optimizations [*Muchnick*, 1997; *Bacon et al.*, 1994; *Alfred V. Aho et al.*, 2007]. For high-level optimizations such as inlining [*Ashley*, 1997] or eliminating run-time type checks [*Jagannathan and Wright*, 1995], however, using flow information is advantageous.

The inlining transformation identifies which call sites call which procedures and decides along the lines of some heuristics whether it is profitable to *inline* the procedure call. That is, the transformation replaces the procedure call with a copy of the code of the procedure being called — specialized for the arguments of the call. If the call site occurs in the body of a loop this transformation is especially beneficial: Without inlining, the cost that accounts for calling the procedure accumulates with every evaluation of the loop body. An inlined call, however, does not produce any overhead for calling a function. Thus, the accumulated cost completely drops out.

To give a concrete example for this transformation, consider the following excerpt of a program:

```
6 (P square_8 (c_2 x_7)
8   (LET* (((v_25)* (* x_7 x_7)))
      (unknown-return 0 c_2 v_25)))

10 (P compute_10 (c_20 v_15 n_3)
12   (LET* (((v_24)* (- v_15 n_3))
14          ((v_27)  (call square_8 v_24)))
       (unknown-return c_20 v_27)))
```

The procedure `compute_10` calls `square_8` to compute the square of a number (in the body of continuation `c_12`). Suppose the compiler wants to inline this call. The first step involves deriving a version of `square_8` specialized for the arguments at this particular call site. In this case, this implies replacing the parameter `c_2` with the argument `c_20` and `x_7` with `v_24`. The result looks like this:

```
6 (P square_8 (c_2 x_7)
8     (LET* (((v_25)* (* v_24 v_24)))
        (unknown-return c_20 v_25)))
```

In a second step, the inlining transformation replaces the call with the body of the specialized procedure:

```
10 (P compute_10 (c_20 v_15 n_3)
12   (LET* (((v_24)* (- v_15 n_3))
14           ((v_25)  (* v_24 v_24)))
       (unknown-return c_20 v_25)))
```

The benefits of flow analysis for an inlining transformation are twofold: First, the control-flow analysis determines the targets of all calls and thus widens the range of opportunities to inline procedures [*Ashley*, 1997]. Second, the flow information helps to find the most promising opportunities among all inlining opportunities.

Consider Figure 3.1 again. The flow information is helpful in both aspects here. The call in the body of continuation `c_29` is an unknown call. That is, the compiler was not able to determine the target of the call. By looking at the flow graph, however, it is apparent that the call targets `square_8`. Here, the flow analysis has discovered a new opportunity to inline.

Among the identified chances to inline, the inlining transformation chooses the candidate that promises the biggest speed-up. Here, knowing if a call takes place inside a loop is a valuable hint for the selection process. Again, a look at the flow graph reveals this information.

3.3 Control-flow analysis for higher-order languages

Computing the control-flow information for programs in a higher-order language such as the intermediate language of the transformational compiler is a well-studied problem. A first approach to this problem goes back to Shivers [*Shivers*, 1991] who uses an abstract interpretation [*Cousot and Cousot*, 1977] to compute the control and data flow. In his thesis, Shivers shows how to construct such an abstract interpreter systematically and proves that the abstract interpreter is a correct simulation of a standard interpreter. This section introduces the idea of control-flow analysis by abstract interpretation informally using a small example.

Consider the flow graph shown in Figure 3.1 again. Note that the compiler could not identify the procedure being called at `(unknown-call p_24 i_23)`. The expression in operator position is `p_24` — a variable. To determine the target of this call, the closure bound to `p_24` must be known.

A control-flow analysis traces the values of variables and therefore identifies the target of the call. The mode of operation is similar to a standard interpreter that evaluates the program. Consider the evaluation in a standard interpreter first. The state of the interpreter consists of an expression to evaluate and an environment that maps a variable to its value. Evaluation in the example starts with `main_46` and reaches the `jump` call in the body of `letrec`. At this time the environment in the interpreter has the following entries: The entry for `v_19` maps to a vector and `lp_21` maps to a closure created over `lp_25`. The interpreter now evaluates the `jump` call and control moves to the procedure `lp_25` (the variable `lp_21` in operator expressions evaluates to the closure over `lp_25`).

The evaluation of this call adds two entries to the variable environment to bind the arguments `i_23` and `p_24` of `lp_24`: `i_23` maps to the integer zero and

`p_24` maps to the closure created over `square_8`. If the control now proceeds to continuation `c_29` and the call `(unknown-call p_24 i_23)`, the evaluation of `p_24` yields the closure over `square_8`.

The control-flow analysis is an abstract interpreter that simulates the standard interpreter. The abstract interpreter differs from the standard interpreter in the following aspects:

- The evaluation process always terminates. The abstract interpreter only distinguishes a finite number of states. For example, it collapses all calls to the loop procedure `lp_25` into a single state. That is, a single state in the abstract interpretation may simulate many states in a real program run.
- The abstract interpreter adds each state visited during the evaluation to the so-called *visited set*.
- Values are simulated by sets of abstract values.

In the end, the visited set contains the results of the simulation. For the example above, the final visited set includes a state that simulates the evaluation of the call `(unknown-call p_24 i_23)`. This state contains a simulation for the environment. Looking up `p_24` in this environment then yields an abstract value that simulates the closure over `square_8`. The control-flow analysis identified the target of the call.

Chapter 4

Analysis Framework

The formal semantics defined in Chapter 2 is the foundation for the flow analysis: Abstracting and instrumenting these semantics yields the *flow-analysis semantics*. This semantics forms a precise description for obtaining the flow information. An interpreter that implements this semantics then carries out the task of collecting the flow information.

The abstracted semantics obeys two properties: First, the semantics is computable. That is, for any given program, including programs that contain infinite loops, the analysis terminates and delivers flow information. Second, the flow information computed is semantically correct.

In the course of this chapter, *concrete semantics* refers to the programming language semantics of the intermediate language as defined in Chapter 2. Flow-analysis semantics, accordingly, is also termed *abstract semantics*. Each semantic domain in the abstract semantics has a counterpart in the concrete semantics. This dissertation follows the widely-used convention of distinguishing both domains using a hat: $\mathbf{Proc}$ denotes the domain of procedure values in the concrete semantics while $\widehat{\mathbf{Proc}}$ denotes the domain of procedure values in the abstract semantics.

This chapter contains an explanation of the flow analysis semantics: The semantic domains (Section 4.1), the transition rules (Section 4.2), and the semantic correctness (Section 4.3). In preparation to analyze realistic programs, Sections 4.4.1 and 4.4.2 extend the abstraction functions for PreScheme and Scheme values and primops. Section 4.5 discusses the choice of abstraction functions made in this chapter and identifies opportunities to render the analysis more precise. To clarify the mode of operation of the analysis, Section 4.6 presents the flow information for two small example programs. Section 5 develops an executable model of the semantics used to trace and inspect the semantics.

4.1 Semantic domains

The evaluation of a program under the concrete semantics may produce an infinite number of states. A program that contains an infinite loop, for example, would expose this property. Thus, the semantics is uncomputable. Flow analyses, on the other hand, are practical tools for a compiler and hence must be

computable for all input programs. The key idea for deriving a computable version of the semantics is to keep the set of states finite. If the abstract semantics ensures that a program can only produce a finite set of states the analysis is computable.

An abstract state typically has more than one concrete counterparts. This is the key idea and expands to other domains as well. This section describes how to abstract the set of values, states and other entities in the concrete semantics to finite sets. Of course it is essential that the abstractions are meaningful. Section 4.3 therefore studies the semantic correctness of the abstractions defined in this section.

Consider the domains for the concrete semantics from as already seen in Figure 2.2:

$$
\begin{array}{llll}
\varsigma \in & \mathbf{State} & = & \mathbf{Eval} + \mathbf{Apply} \\
& \mathbf{Eval} & = & \mathtt{Call} \times \mathbf{BEnv} \times \mathbf{VEnv} \times \mathbf{Store} \times \mathbf{Time} \\
& \mathbf{Apply} & = & \mathbf{Proc} \times \mathbf{D}^* \times \mathbf{D}^* \times \mathbf{VEnv} \times \mathbf{Store} \times \mathbf{Time} \\
\beta \in & \mathbf{BEnv} & = & \mathtt{Var} \to \mathbf{Time} \\
ve \in & \mathbf{VEnv} & = & \mathtt{Var} \times \mathbf{Time} \to \mathbf{D} \\
& \mathbf{HLoc} & = & \mathtt{Lab} \times \mathbf{Time} \\
& \mathbf{GLoc} & = & \mathtt{Var} \\
loc \in & \mathbf{Loc} & = & \mathbf{HLoc} + \mathbf{GLoc} \\
ref \in & \mathbf{Ref} & = & \mathbf{Loc} \times \mathtt{Lit}_{Lang} \\
\sigma \in & \mathbf{Store} & = & \mathbf{Ref} \to \mathbf{D} \\
proc \in & \mathbf{Proc} & = & \mathbf{Clo} + \{\mathtt{halt}\} \\
clos \in & \mathbf{Clo} & = & \mathtt{Lam} \times \mathbf{BEnv} \times \mathbf{Time} \\
& \mathbf{D} & = & \mathbf{Proc} + \mathbf{Ref} + \mathbf{Bas}_{Lang} + \mathbf{Comp}_{Lang} \\
t \in & \mathbf{Time} & = & \mathbb{N}
\end{array}
$$

State is infinite if at least one of the component domains of a state has an infinite number of elements. **Time** has an infinite number of elements because the concrete semantics encode time using natural numbers.

So the first step towards a finite set of states is to define a finite set of abstract points in time. For the time being, this informal definition of $\widehat{\mathbf{Time}}$ is sufficient:

$$\hat{t} \in \widehat{\mathbf{Time}} = \text{finite set of abstract time}$$

The choice of $\widehat{\mathbf{Time}}$ is the most important adjusting screw for controlling the precision of the analysis. Section 4.5 revisits this subject in depth. $\widehat{tick}$ is the abstract version of *tick*. Note that $\widehat{tick}$ in contrast to *tick* has a second argument: an *eval* state. The k-CFA time abstractions discussed in Section 4.5 uses the state to extract context information and distinguish multiple dynamic instances of a call. Both versions of $\widehat{tick}$ follow this declaration:

$$\widehat{tick} : (\widehat{\mathbf{Time}} \times \widehat{\mathbf{Eval}}) \to \widehat{\mathbf{Time}}$$

Figure 4.1 lists all abstract semantic domains of the flow analysis. With the exceptions of abstract values $\widehat{\mathbf{D}}$ and $\widehat{\mathbf{Time}}$ these domains match their concrete counterparts. When time abstracts to a finite set of times, the other components of a state also become finite:

- Binding environments **BEnv** maps a finite set of variables to a finite set of points in time.

$$
\begin{array}{llll}
\widehat{\varsigma} & \in & \widehat{\mathbf{State}} & = \widehat{\mathbf{Eval}} + \widehat{\mathbf{Apply}} \\
 & & \widehat{\mathbf{Eval}} & = \mathtt{Call} \times \widehat{\mathbf{BEnv}} \times \widehat{\mathbf{VEnv}} \times \widehat{\mathbf{Store}} \times \widehat{\mathbf{Time}} \\
 & & \widehat{\mathbf{Apply}} & = \widehat{\mathbf{Proc}} \times \widehat{\mathbf{D}}^* \times \widehat{\mathbf{D}}^* \times \widehat{\mathbf{VEnv}} \times \widehat{\mathbf{Store}} \times \widehat{\mathbf{Time}} \\
\widehat{\beta} & \in & \widehat{\mathbf{BEnv}} & = \mathtt{Var} \to \widehat{\mathbf{Time}} \\
\widehat{ve} & \in & \widehat{\mathbf{VEnv}} & = \mathtt{Var} \times \widehat{\mathbf{Time}} \to \widehat{\mathbf{D}} \\
 & & \widehat{\mathbf{HLoc}} & = \mathtt{Lab} \times \widehat{\mathbf{Time}} \\
 & & \widehat{\mathbf{GLoc}} & = \mathtt{Var} \\
\widehat{loc} & \in & \widehat{\mathbf{Loc}} & = \widehat{\mathbf{HLoc}} + \widehat{\mathbf{GLoc}} \\
\widehat{ref} & \in & \widehat{\mathbf{Ref}} & = \widehat{\mathbf{Loc}} \times \mathtt{Lit}_{Lang} \\
\widehat{\sigma} & \in & \widehat{\mathbf{Store}} & = \widehat{\mathbf{Ref}} \to \mathbf{D} \\
\widehat{proc} & \in & \widehat{\mathbf{Proc}} & = \widehat{\mathbf{Clo}} + \{\mathtt{halt}\} \\
\widehat{clos} & \in & \widehat{\mathbf{Clo}} & = \mathtt{Lam} \times \widehat{\mathbf{BEnv}} \times \widehat{\mathbf{Time}} \\
 & & \widehat{\mathbf{D}} & = \mathcal{P}(\widehat{\mathbf{Proc}} + \widehat{\mathbf{Ref}} + \widehat{\mathbf{Bas}}_{Lang} + \widehat{\mathbf{Comp}}_{Lang}) \\
\widehat{t} & \in & \widehat{\mathbf{Time}} & = \text{finite set of abstract times}
\end{array}
$$

Figure 4.1: Flow analysis semantic domains

- A closure consists of a lambda form, a binding environment, and a point in time all of which are finite. Hence, **Clo** and **Proc** are also finite.
- As $\widehat{\mathbf{Bas}}_{Lang}$ and $\widehat{\mathbf{Comp}}_{Lang}$ are finite, $\widehat{\mathbf{D}}$ is as well.
- The set of references $\widehat{\mathbf{Ref}}$ is finite because the number of literals and variables in a program of finite length must also be finite. There are also only a finite number of heap locations $\widehat{\mathbf{HLoc}}$ as the number of call nodes in a program are finite and $\widehat{\mathbf{Time}}$ is also finite.
- Since there is a finite set of references, the number of entries in $\widehat{\mathbf{Store}}$ is also finite.
- As $\widehat{\mathbf{D}}$ is finite the variable environment **VEnv** is finite: `Var` is finite, $\widehat{\mathbf{Time}}$ is by definition finite, and the set of abstract values $\widehat{\mathbf{D}}$ is finite as well.

Consequently, $\widehat{\mathbf{State}}$ is finite.

In the abstract semantics denotable values are represented by sets: $\widehat{\mathbf{D}}$ is now the powerset over the summands of the denotable value domain.

Denotable values, internal run-time values not available to the program such as environments, and syntactic categories build up the semantic domains of the analysis. So far, these domains are unordered sets. The flow analysis, however, is a fixed-point computation. Hence, constructing a lattice over the semantic domains is a prerequisite.

Extending the domains to lattices is a two-step process. First, the relation $\sqsubseteq$ induces a partial ordering on the elements of the set. The second step adds elements for the top and the bottom elements to the sets. This ensures that all subsets have a least and greatest element. Taken together, these steps establish the lattice property [*Mitchell*, 1996; *Burris and Sankappanavar*, 1999]. The

symbols $\sqcap$ and $\sqcup$ denote the meet and join operators of these lattices. Join, meet and $\sqsubseteq$ for the domain types are defined as follows:

Sets Extending simple sets, such as the syntactic domains, to lattices is straightforward. In theses cases $\sqsubseteq$ is an equivalence relation for this set. Sets contain a special least and greatest element $\bot$ and $\top$ unless the set is a singleton set. The usual intersection $\cap$ and union $\cup$ set operations constitute meet and join.

Power lattice Let A be a power lattice over a lattice B defined as $A = \mathcal{P}(B)$. Least and greatest elements are defined as:

$$\bot_A = \varnothing \qquad \top_A = B$$

Comparing elements of power lattices involves using the proper order relation $\sqsubseteq_B$. The definition for $\sqsubseteq_B$ is:

$$A_1 \sqsubseteq_A A_2 \quad = \quad \forall b_1 \in A_1 : \ \exists b_2 \in A_2 : \ b_1 \sqsubseteq_B b_2$$

$\sqcup_A$ for power domains is defined as the union of sets. The meet operation, however, is more complex. Again, using the proper order relation is necessary:

$$A_1 \sqcap_A A_2 \quad = \quad \{\{b\} \in (A_1 \cap A_2) : \ \{b\} \sqsubseteq_A A_1 \ \wedge \ \{b\} \sqsubseteq_A A_2\}$$

Product lattice Let C be a product lattice defined as $C = A \times B$. The least and greatest elements are defined as follows:

$$\bot_C = (\bot_A, \bot_B) \qquad \top_C = (\top_A, \top_B)$$

For product lattices, $\sqsubseteq$ simply extends to the components:

$$(a_1, b_1) \sqsubseteq_C (a_2, b_2) \quad \text{iff} \quad a_1 \sqsubseteq_A a_2 \ \wedge \ b_1 \sqsubseteq_B b_2$$

The join and meet operations work component-wise and are defined as follows:

$$(a_1, b_1) \sqcap_C (a_2, b_2) = (a_1 \sqcap_A a_2, \ b_1 \sqcap_B b_2)$$
$$(a_1, b_1) \sqcup_C (a_2, b_2) = (a_1 \sqcup_A a_2, \ b_1 \sqcup_B b_2)$$

Function lattice Let C be a function lattice defined as $C = A \rightarrow B$. C's least and greatest elements are defined as:

$$\bot_C = \lambda a.\, \bot_B \qquad \top_C = \lambda a.\, \top_B$$

The order relation $\sqsubseteq_C$ works element-wise:

$$f \sqsubseteq_C g \quad \text{iff} \quad \forall a \in \text{dom}(f) : \ f(a) \sqsubseteq_B g(a)$$

Join and meet lift the operation to the function domain and are defined as:

$$f \sqcup_C g = \lambda a.\, (f(a) \sqcup_B g(a))$$
$$f \sqcap_C g = \lambda a.\, (f(a) \sqcap_C g(a))$$

Vector lattice Let A be the vector lattice defined as $A = B^*$. Then least and greatest elements are defined as follows:

$$\bot_A = \langle \bot_B, \ldots, \bot_B \rangle$$
$$\top_A = \langle \top_B, \ldots, \top_B \rangle$$

The relation $\sqsubseteq_A$ works element-wise and is defined as:

$$\langle u_1, \ldots, u_n \rangle \sqsubseteq_A \langle v_1, \ldots, v_n \rangle \quad \text{iff} \quad \forall i : \ 1 \leq i \leq n : \ u_i \sqsubseteq_B v_i$$

Join and meet also work element-wise and are defined as:

$$\langle u_1, \ldots, u_n \rangle \sqcup_A \langle v_1, \ldots, v_n \rangle = \langle u_1 \sqcup_B v_1, \ldots, u_n \sqcup_B v_n \rangle$$
$$\langle u_1, \ldots, u_n \rangle \sqcap_A \langle v_1, \ldots, v_n \rangle = \langle u_1 \sqcap_B v_1, \ldots, u_n \sqcap_B v_n \rangle$$

Sum lattice Let C be a sum lattice defined as $C = A + B$. Adding new least and greatest elements $\bot_C$ and $\top_C$ is necessary. The relation $\sqsubseteq_C$ is defined as:

$$u \sqsubseteq_C v = \begin{cases} u \sqsubseteq_A v & u \in A \text{ and } v \in A \\ u \sqsubseteq_B v & u \in B \text{ and } v \in B \\ \mathit{true} & u = \bot_C \\ \mathit{true} & v = \top_C \\ \mathit{false} & \text{otherwise} \end{cases}$$

Join and meet on sum lattices are defined as:

$$u \sqcup_C v = \begin{cases} u \sqcup_A v & u \in A \text{ and } v \in A \\ u \sqcup_B v & u \in B \text{ and } v \in B \\ \top_C & \text{otherwise} \end{cases}$$

$$u \sqcap_C v = \begin{cases} u \sqcap_A v & u \in A \text{ and } v \in A \\ u \sqcap_B v & u \in B \text{ and } v \in B \\ \bot_C & \text{otherwise} \end{cases}$$

Lemma 1. *The relation $\sqsubseteq$ is reflexive, transitive, and anti-symmetric.*

Proof. Follows immediately from the definitions above. □

The above definitions for $\sqsubseteq$, join, and meet operations complete the definition of the semantic domains for the flow semantics.

4.2 State transition

This section defines the state transition relation $\widehat{\Rightarrow}$ of the abstract semantics. The transitions of the concrete semantics serve as a guideline (see Section 2.3). It is important to understand that, unlike in the concrete case, a state transition may yield more than just one successor state.

Figure 4.2 shows the transition rules for *eval* states. Consider $\widehat{\text{PCallEval}}$ first. The basic mode of operation is the same as in the concrete case: $\widehat{\mathcal{A}}$ evaluates the operator, operands, and continuation of the call under the current environments.

$$\frac{prim_p \in \texttt{Prim}_{PCall} \setminus \{\texttt{letrec1}, \texttt{letrec2}\}}{\widehat{\varsigma} = ((prim_p\ \langle c, f, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{p_i}\}, \langle \widehat{c'} \rangle, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}} \quad (\widehat{\text{PCallEval}})$$

$$\text{where} \begin{cases} \widehat{p_i} & \in \widehat{proc} \\ \widehat{proc} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ f \\ \widehat{c'} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ \widehat{d_i} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ a_i \\ \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \end{cases}$$

$$\frac{prim_c \in \texttt{Prim}_{CCall} \setminus \{\texttt{test}\}}{\widehat{\varsigma} = ((prim_c\ \langle c, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{p_i}\}, \langle\rangle, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}} \quad (\widehat{\text{CCallEval}})$$

$$\text{where} \begin{cases} \widehat{p_i} & \in \widehat{proc} \\ \widehat{proc} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ \widehat{d_i} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ a_i \\ \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \end{cases}$$

$$\frac{}{\widehat{\varsigma} = ((prim_l\ \langle c, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle \widehat{r} \rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t'})\}} \quad (\widehat{\text{PrimCallEval}})$$

$$\text{where} \begin{cases} \widehat{d_i} & \in \widehat{d} \\ \widehat{d} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ (\widehat{r}, \widehat{\sigma}') & = \widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ prim_l \langle a_1, \ldots, a_n\rangle \\ \widehat{t'} & = \widehat{tick}(\widehat{t'}, \widehat{\varsigma}) \end{cases}$$

Figure 4.2: Simple transitions for abstract *eval* states

However, the operator f evaluates to a set of abstract procedure values $\widehat{proc}$ here — the set of abstract closures that may occur during run time of the program as the operators of this call. The transition rule then creates an *apply* state for each abstract procedure value in $\widehat{proc}$: The set of procedure values in an *apply* state is always a singleton set and the states created by one invocation of this rule only differ by this set. In the actual implementation, splitting the *apply* states this way makes searching for states which apply a certain closure easier and faster — an operation which occurs often. The remaining rules creating *apply* states also obey this convention.

Transition rule $\widehat{\text{PrimCallEval}}$ depends on $\widehat{\mathcal{P}}_{Lang}$, the function that evaluates the call to the language-specific primop $prim_l$ and returns an abstract value. Thus, $\widehat{\mathcal{P}}_{Lang}$ is an interface to the language-specific part of the analysis. See Section 4.4 for a complete summary of the interface for language-specific abstractions and examples for $\widehat{\mathcal{P}}_{Lang}$.

The $\widehat{\text{TestEval}}$ rule shown in the upper half of Figure 4.3 evaluates calls using the `test` primop. An application of $\widehat{\text{TestEval}}$ results in two *apply* states, one for

$$\frac{}{\widehat{\varsigma} = (\texttt{(test } \langle c_0, c_1, a\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\widehat{b_i}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}} \quad (\widehat{\text{TestEval}})$$

$$\text{where} \begin{cases} \widehat{r} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ a \\ \widehat{b_0} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c_0 \\ \widehat{b_1} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c_1 \\ \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \end{cases}$$

$$\frac{}{\widehat{\varsigma} = \begin{array}{l}(\texttt{(letrec1 } \langle(\lambda\ (v_1^c \ldots v_n^c) \\ \quad \texttt{(letrec2 } \langle l^c, e_1^u \ldots e_n^u\rangle)_\tau\rangle)_{\tau'})_{\tau''}, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})\end{array} \widehat{\Rightarrow} (call, \widehat{\beta'}, \widehat{ve}', \widehat{\sigma}, \widehat{t'})} \quad (\widehat{\text{LetrecEval}})$$

$$\text{where} \begin{cases} (\lambda^c()\ call) & = l_c \\ \widehat{r_i} & = \widehat{\mathcal{A}}\, \widehat{\beta'}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ e_i^u \\ \widehat{\beta'} & = \widehat{\beta}[v_i \mapsto \widehat{t'}] \\ \widehat{ve}' & = \widehat{ve} \sqcup [(v_i^c, \widehat{t'}) \mapsto \widehat{r_i}] \\ \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \end{cases}$$

Figure 4.3: Complex transitions for abstract *eval* states

each continuation of the `test` primop — this is independent of the abstract value to which the test expression evaluates. The analysis assumes that the language-specific abstract values for booleans are not precise enough to consider only one branch.

The lower half of Figure 4.3 shows the transition rules for calls using the `letrec` primop. Rule $\widehat{\text{LetrecEval}}$ works in the same way as its concrete counterpart LetrecEval.

Figure 4.4 shows the transition rules for the primops `global-ref` and `global-set!`. Since both rules apply to *eval* states, they also advance the time to $\widehat{t'}$. Like its concrete counterpart, the rule $\widehat{\text{PGlobalRef}}$ searches the store for the reference to the global variable g and calls all abstract continuations with the result. Trivial calls may also use `global-ref` as a primop and the following case for $\widehat{\mathcal{P}}_{Lang}$ evaluates these calls:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\, \texttt{global-ref}\ g = \widehat{\sigma}(g, \top)$$

The rule $\widehat{\text{PGlobalSet}}$ updates the store for a global variable g. The rule proceeds in three steps. First, it evaluates the argument a — the new value for g — and the continuation argument of the call. The second step updates the store $\widehat{\sigma}$ by merging the old value for g with its new value computed in the first step. Finally, $\widehat{\text{PGlobalSet}}$ constructs *apply* states for each continuation of the call to `global-set!`.

$\widehat{\mathcal{A}}$ is the abstract counterpart of $\mathcal{A}$, the auxiliary function that evaluates the arguments of a call. Figure 4.5 shows $\widehat{\mathcal{A}}$. The only difference concerns the return values, which are sets of denotable values now. Note that $\widehat{\mathcal{A}}$ depends on the language specific abstraction functions $\widehat{\mathcal{P}}_{Lang}$ and $\widehat{\mathcal{K}}_{Lang}$.

$$\frac{}{(\texttt{(global-ref}\ \langle c, g\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} (\{\widehat{p_i}\}, \langle\widehat{r}\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})} \quad (\widehat{\text{PGlobalRef}})$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}) \\ \widehat{p_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{r} & = \widehat{\sigma}(g, \top) \end{cases}$$

$$\frac{}{(\texttt{(global-set!}\ \langle c, g, a\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} (\{\widehat{p_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t'})} \quad (\widehat{\text{PGlobalSet}})$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}) \\ \widehat{p_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{d} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,a \\ \widehat{\sigma}' & = \widehat{\sigma} \sqcup [(g, \top) \mapsto \widehat{d}] \end{cases}$$

Figure 4.4: Transitions for accessing the abstract store

$$\begin{aligned} \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,lam &= \{(lam, \widehat{\beta}, \widehat{t})\} \\ \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,v &= \widehat{ve}(v, \widehat{\beta}(v)) \\ \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,g &= \widehat{\sigma}(g, \top) \\ \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,lit &= \{\widehat{\mathcal{K}}_{Lang}\,lit\} \\ \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,\texttt{(}prim\ \langle e_1^u, \ldots, e_n^u\rangle\texttt{)} &= \widehat{\mathcal{P}}_{Lang}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,prim\langle e_1^u, \ldots, e_n^u\rangle \end{aligned}$$

Figure 4.5: Evaluating call arguments

Figure 4.6 shows the transition rule $\widehat{\text{ApplyClos}}$: The transition from *apply* to *eval* states. Like its concrete counterpart $\widehat{\text{ApplyClos}}$ binds the abstract values provided in the argument vectors $\widehat{c^*}$ and $\widehat{d^*}$ to the parameters of the lambda expression. Binding the values consists of two stages. First, $\widehat{\text{ApplyClos}}$ creates a binding environment that maps the parameters to the current time. The second step updates the variable environment by merging the new values into the original variable environment. Note that the concrete counterpart of $\widehat{\text{ApplyClos}}$ actually updates the variable environment — that is, replaces the old values by the new values. The successor state is an *eval* state which evaluates the call in the lambda expression's body under the updated environments.

The $\widehat{\text{ApplyClos}}$ rule completes the state transition rules. Now, almost everything is set up for defining the flow analysis of a program. The flow analysis is a set containing all states visited during the computation of $\widehat{\Rightarrow}^*$ for a given program pr. This set of states is the *visited set* $\widehat{\mathcal{V}}$:

$$\widehat{\mathcal{V}}(pr) = \{\widehat{\varsigma}_I \widehat{\Rightarrow}^* \widehat{\varsigma}\} \qquad \widehat{\varsigma}_I \in \widehat{\mathcal{I}}(pr)$$

$$\frac{\text{length}(\widehat{c}^* \S \widehat{d}^*) = n}{(\{((\lambda\ (p_1 \ldots p_n)\ call)_\tau, \widehat{\beta}, \widehat{t}_b)\}, \widehat{c}^*, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} (call, \widehat{\beta}', \widehat{ve}', \widehat{\sigma}, \widehat{t})}\ (\widehat{\text{ApplyClos}})$$

$$\text{where} \begin{cases} \widehat{\beta}' & = \widehat{\beta}[p_i \mapsto \widehat{t}] \\ \widehat{ve}' & = \widehat{ve} \sqcup [(p_i, \widehat{t}) \mapsto (\widehat{c}^* \S \widehat{d}^*)_i] \end{cases}$$

Figure 4.6: Transition for abstract *apply* states

The function $\widehat{\mathcal{I}}$ is the abstract counterpart of $\mathcal{I}$ and computes the initial abstract state of a program:

$$\widehat{\mathcal{I}} : \texttt{Prog} \to \widehat{\textbf{State}}$$

The functions $\widehat{\mathcal{I}}$ and $\mathcal{I}$ are very similar. The initial state is an *apply* state that applies l_{main} to the `halt` continuation. The initial binding environment $\widehat{\beta}_0$ and the initial variable environment $\widehat{ve}_0$ are empty. The initial store $\widehat{\sigma}_I$ contains references to all globally defined values in the program *pr*:

$$\widehat{\mathcal{I}}(\langle f^*, l_{main} \rangle) = \langle \{\langle l_{main}, \widehat{\beta}_0, \widehat{t}_0 \rangle\}, \langle\rangle, \langle \{\texttt{halt}\} \rangle, \widehat{ve}_0, \widehat{\sigma}_I, \widehat{t}_0 \rangle$$

Section 2.4.3 explains how the concrete semantics constructs the initial store. Constructing the abstract counterpart uses abstract domains and the abstract argument evaluation function:

$$\begin{aligned} \widehat{\sigma}_0 &= [] \\ \widehat{\sigma}_i &= \widehat{\sigma}_{i-1}[(g_i, \top) \mapsto \widehat{\mathcal{A}}\, \widehat{\beta}_0\, \widehat{ve}_0\, \widehat{\sigma}_{i-1}\, e_i\,] \\ \widehat{\sigma}_I &= \widehat{\sigma}_n \\ & \quad \text{where} \quad \forall (g_i, e_i) = f_i^*, f^* \in \texttt{Definition}^* \end{aligned}$$

As in the concrete semantics, the forms of the program must obey the `letrec*` property as stated in Section 2.4.3.

The abstract semantics just defined establishes the foundation for a practical implementation of the flow analysis in a compiler: Computing the flow analysis means computing the visited set. To make this more concrete, Figure 4.7 shows pseudocode for actually computing the analysis. This pseudocode only serves as an aid in understanding how the analysis works: Chapter 6 discusses the implementation aspects in depth. *prog* in the pseudocode denotes the abstract syntax tree of the input program.

The algorithm is a work-list algorithm. The central data structures are an ordinary first-in, first-out queue that holds the states to visit (named `unvisited` in the code) and the set of visited states: `visited`. In a first step, the program fills `unvisited` with the initial state computed by $\widehat{\mathcal{I}}$. The program then enters a loop which runs as long as the queue contains states to visit. If the queue contains a state, the algorithm removes this state from the queue, includes it in the visited set and computes its successor states. The successor states may either be new, unvisited states or states the analysis already discovered. The algorithm adds only unvisited states to the queue and therefore checks whether the unvisited queue or visited set contain successor state in question. Here, it is important to use the relation $\sqsubseteq$ to compare states: $\sqsubseteq$ is called the *approximation*

```
(define (analyze prog)
  (let ((initial-states Î(prog))
        (unvisited (make-queue)))
    (for-each (lambda (s) (enqueue! unvisited s))
              initial-states)
    (let lp ((visited ∅))
      (if (queue-empty? unvisited)
          visited
          (let* ((s (dequeue! unvisited))
                 (new-visited visited ∪ {s})
                 (successors (s ⇀̂)))
            (for-each
              (lambda (ns)
                (if (not (and (find-in-set ns new-visited ⊑)
                              (find-in-queue ns unvisited ⊑)))
                    (enqueue! unvisited ns)))
              successors)
            (lp new-visited))))))
```

Figure 4.7: Pseudocode for computing the analysis

relation. Consider two abstract states $\widehat{u}$ and $\widehat{v}$ for which $\widehat{u} \sqsubseteq \widehat{v}$ holds. Then, $\widehat{v}$ is said to approximate $\widehat{u}$. That is, if $\widehat{u}$ is a *valid abstraction* (see the next section for a formal definition) of some concrete state u, then $\widehat{v}$ is also a valid abstraction of u (see Theorem 2 in Appendix B). Note that $\sqsubseteq$ is defined for all entities in the flow analysis.

Searching the visited set for a state that approximates the state just created is called the *termination check*. This check ensures that only states that contain new information — states, which are not approximated by any state in the visited set — are scheduled for a further state transition. Hence, this check prevents that the analysis considers the same information over and over again.

In a practical setting the termination check dominates the run time of the analysis. Therefore, the termination check is a crucial to the analysis implementation (see Chapter 6). The algorithm terminates when the `unvisited` queue is empty and returns the set of visited states.

4.3 Semantic correctness

So far the concrete semantics has served as a template for developing the abstract semantics. Recall that the abstract semantics must obey two properties: The semantics must be computable and the flow-analysis results must be a simulation of all actual program runs. The abstract semantics is computable: The set of abstract states is finite.

This section addresses the second property — the semantic correctness of the flow analysis. As discussed in Section 3, the results of a flow analysis is useful for high-level program optimizations. These optimizations draw conclusions on the run-time behavior of a program on the basis of flow-analysis results. Hence

it is necessary to establish a formal understanding how the flow-analysis results relate to the run-time behavior of a program.

In preparation for the proof, a formal definition for the run-time behavior according to the concrete semantics is necessary: The set of states visited during the evaluation defines the behavior. For a program pr the visited set $\mathcal{V}(pr)$ is defined by:

$$\mathcal{V}(pr) = \{\varsigma_i \rightharpoonup^* \varsigma\} \qquad \varsigma_i \in \mathcal{I}(pr)$$

For formulating the central semantic correctness theorem, one further ingredient is necessary: A relation that connects concrete states and values with their abstract counterparts:

$$\mathcal{R}_{State} : \mathbf{State} \times \widehat{\mathbf{State}} \rightarrow \{true, false\}$$

Given a concrete and an abstract state, $\mathcal{R}_{State}$ computes whether the abstract state is an abstraction of the concrete state. Essentially, semantic correctness means semantic correctness relative to connection defined by $\mathcal{R}_{State}$ — it is the *correctness relation* of the flow analysis. Basically, relating states yields to checking the relation for the state components. Hence, this section defines correctness relations for abstract values and environments. All the relations have the name $\mathcal{R}$ with a subscript indicating what type of objects it relates. Where it is non-ambiguous the subscript may be left out.

Consider the definition of $\mathcal{R}_{State}$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\texttt{(}prim\ \langle e_1, \ldots, e_n\rangle\texttt{)}_\tau, \beta, ve, \sigma, t)\ \mathcal{R}_{State}\ (\texttt{(}prim\ \langle e_1, \ldots, e_n\rangle\texttt{)}_{\tau'}, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \quad \text{iff}} \\
\quad \tau = \tau' & \wedge \\
\quad \beta\ \mathcal{R}_{BEnv}\ \widehat{\beta} & \wedge \\
\quad ve\ \mathcal{R}_{VEnv}\ \widehat{ve} & \wedge \\
\quad \sigma\ \mathcal{R}_{Store}\ \widehat{\sigma} & \wedge \\
\quad t\ \mathcal{R}_{Time}\ \widehat{t} & \\
\multicolumn{2}{l}{(proc, d^*, c^*, ve, \sigma, t)\ \mathcal{R}_{State}\ (\{\widehat{proc}\}, \widehat{d^*}, \widehat{c^*}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \quad \text{iff}} \\
\quad proc\ \mathcal{R}_{Proc}\ \widehat{proc} & \wedge \\
\quad d^*\ \mathcal{R}_{D^*}\ \widehat{d^*} & \wedge \\
\quad c^*\ \mathcal{R}_{D^*}\ \widehat{c^*} & \wedge \\
\quad ve\ \mathcal{R}_{VEnv}\ \widehat{ve} & \wedge \\
\quad \sigma\ \mathcal{R}_{Store}\ \widehat{\sigma} & \wedge \\
\quad t\ \mathcal{R}_{Time}\ \widehat{t} &
\end{array}$$

This definition extends $\mathcal{R}$ on the components of a state: An abstract *eval* state corresponds to a concrete *eval* state if both states concern the same call expression and if both environments are abstractions over their concrete counterparts. For *apply* states, the situation is similar: However, in addition checking the variable environment it is necessary to determine whether the argument vector and the operator is a proper abstraction for *proc*.

For each domain $\widehat{X}$ the top value $\top_{\widehat{X}}$ of the abstract domain represents all values from the concrete counterpart:

$$x\ \mathcal{R}_X\ \top_{\widehat{X}} = true \qquad x \in X$$

Elements of **Proc** are either the special `halt` continuation or closures. Hence, the definition of $\mathcal{R}_{Proc}$ distinguishes two cases:

$$\mathcal{R}_{Proc} : \mathbf{Proc} \times \widehat{\mathbf{Proc}} \to \{\mathit{true}, \mathit{false}\}$$

$$\begin{array}{l}
\texttt{halt}\ \mathcal{R}_{Proc}\ \texttt{halt} = \mathit{true} \\
((\lambda(v_1 \ldots v_n)\ \mathit{call})_\tau, \beta, t)\ \mathcal{R}_{Proc}\ ((\lambda(v_1 \ldots v_n)\ \mathit{call})_{\tau'}, \widehat{\beta}, \widehat{t})\ \text{ iff} \\
\qquad \tau = \tau' \quad \wedge \\
\qquad \beta\ \mathcal{R}_{BEnv}\ \widehat{\beta}\ \wedge \\
\qquad t\ \mathcal{R}_{Time}\ \widehat{t}
\end{array}$$

The case for the halt continuation is easy: In both lattices there is exactly one value for `halt` and $\mathcal{R}_{Proc}$ relates them. Moreover, $\mathcal{R}_{Proc}$ relates closures from the same lambda expression. $\mathcal{R}_{Proc}$ includes the binding environments in the check. This allows to distinguish multiple closures created over the same lambda term at different points in time.

Denotable values abstract to a set of abstract values. Thus, $\mathcal{R}_D$ requires that the set of abstract values contains a representative for the concrete value:

$$\mathcal{R}_D : \mathbf{D} \times \widehat{\mathbf{D}} \to \{\mathit{true}, \mathit{false}\}$$

$$\begin{array}{llll}
\mathit{proc}\ \mathcal{R}_D\ \widehat{d} & \text{iff} & \exists \widehat{v} \in \widehat{d} :\ \mathit{proc}\ \mathcal{R}_{Proc}\ \widehat{v} & \\
\mathit{ref}\ \mathcal{R}_D\ \widehat{d} & \text{iff} & \exists \widehat{\mathit{ref}} \in \widehat{d} :\ \mathit{ref}\ \mathcal{R}_{Ref}\ \widehat{\mathit{ref}} & \mathit{ref} \in \mathbf{Ref}, \widehat{\mathit{ref}} \in \widehat{\mathbf{Ref}} \\
b\ \mathcal{R}_D\ \widehat{d} & \text{iff} & \exists \widehat{v} \in \widehat{d} :\ b\ \mathcal{R}_{BVal}\ \widehat{v} & b \in \mathbf{Bas}_{Lang}, \widehat{d} \in \widehat{\mathbf{Bas}}_{Lang} \\
c\ \mathcal{R}_D\ \widehat{d} & \text{iff} & \exists \widehat{v} \in \widehat{d} :\ c\ \mathcal{R}_{CVal}\ \widehat{v} & b \in \mathbf{Comp}_{Lang}, \widehat{b} \in \widehat{\mathbf{Comp}}_{Lang}
\end{array}$$

Basic and compound values are not part of the concrete or abstract semantics because they are specific to the input language. Thus, the rest of this chapter assumes that corresponding definitions of $\mathcal{R}$ for these value types exist.

An abstract reference simulates a concrete reference if the selector matches and if the abstract location models the concrete location:

$$\mathcal{R}_{Ref} : \mathbf{Ref} \times \widehat{\mathbf{Ref}} \to \{\mathit{true}, \mathit{false}\}$$

$$\begin{array}{l}
(\mathit{loc}, \mathit{lit}_1)\ \mathcal{R}_{Ref}\ (\widehat{\mathit{loc}}, \mathit{lit}_2)\ \text{iff} \\
\qquad \mathit{lit}_1 = \mathit{lit}_2 \quad \wedge \\
\qquad \mathit{loc}\ \mathcal{R}_{Loc}\ \widehat{\mathit{loc}}
\end{array}$$

For locations, the correctness relation distinguishes between heap locations and global variable locations. The correctness for global locations goes back to comparing the global variable names. For heap locations, $\mathcal{R}_{Loc}$ checks if the call nodes are identical and if the concrete time models the abstract time:

$$\mathcal{R}_{Loc} : \mathbf{Loc} \times \widehat{\mathbf{Loc}} \to \{\mathit{true}, \mathit{false}\}$$

$$\begin{array}{l}
g_1\ \mathcal{R}_{GLoc}\ g_2\ \text{iff} \\
\qquad g_1 = g_2 \\
(\tau, t)\ \mathcal{R}_{HLoc}\ (\tau', \widehat{t})\ \text{iff} \\
\qquad \tau = \tau' \quad \wedge \\
\qquad t\ \mathcal{R}_{Time}\ \widehat{t}
\end{array}$$

Equipped with the correctness relation for references, locations, and denotable values defining the correctness relation for stores becomes possible. The

correctness relation for stores is analogous to the correctness relation for variable environments. For each entry in the concrete store there must be an entry in the abstract store for which the abstract denotable values simulate the concrete value:

$$\mathcal{R}_{Store} : \mathbf{Store} \times \widehat{\mathbf{Store}} \to \{true, false\}$$

$$\sigma \ \mathcal{R}_{Store} \ \widehat{\sigma} \text{ iff } \forall ref \in \text{dom}(\sigma) : \forall \widehat{ref} \in \widehat{\mathbf{Ref}} : ref \ \mathcal{R}_{Ref} \ \widehat{ref} \Rightarrow \sigma(ref) \ \mathcal{R}_D \ \widehat{\sigma}(\widehat{ref})$$

Arguments to a call are vectors of denotable values and $\mathcal{R}_{D*}$ extends $\mathcal{R}_D$ to the vector elements:

$$\mathcal{R}_{D^*} : \mathbf{D}^* \times \widehat{\mathbf{D}}^* \to \{true, false\}$$

$$\langle d_1, \ldots, d_n \rangle \ \mathcal{R}_{D^*} \ \langle \widehat{d_1}, \ldots, \widehat{d_n} \rangle \quad \text{iff} \quad \forall 0 \leq i \leq n : \ d_i \ \mathcal{R}_D \ \widehat{d_i} = true$$

$\mathcal{R}_{BEnv}$ decides whether an abstract binding environment is an abstraction of a given concrete environment. This requires checking whether the abstract binding environment closes over the same variables as its concrete counterpart and checking whether the time stamps are related:

$$\mathcal{R}_{BEnv} : \mathbf{BEnv} \times \widehat{\mathbf{BEnv}} \to \{true, false\}$$

$$\beta \ \mathcal{R}_{BEnv} \ \widehat{\beta} \quad \text{iff} \quad \forall v \in \text{dom}(\beta) : \ \beta(v) \ \mathcal{R}_{Time} \ \widehat{\beta}(v)$$

A variable environment maps tuples of variables and time stamps to denotable values. In principle the correctness relation checks if for all entries the denotable values relate according to $\mathcal{R}_D$, i. e. if the abstract variable environment binds the variable to suitable representatives of the concrete values. However, in abstract variable environments, time stamps are abstract points in time. Thus, multiple entries in the concrete environment may collapse into a single entry in the abstract environment. Therefore, $\mathcal{R}_{VEnv}$ compares all abstract entries where concrete and abstract time are related:

$$\mathcal{R}_{VEnv} : \mathbf{VEnv} \times \widehat{\mathbf{VEnv}} \to \{true, false\}$$

$$ve \ \mathcal{R}_{VEnv} \ \widehat{ve} \quad \text{iff} \quad \forall (v,t) \in \text{dom}(ve) : \forall \widehat{t} \in \widehat{\mathbf{Time}} : t \ \mathcal{R} \ \widehat{t} \Rightarrow ve(v,t) \ \mathcal{R}_D \ \widehat{ve}(v, \widehat{t})$$

The flow analysis as presented is not tied to a specific time abstraction: various implementations with varying resolutions exists (see Section 4.5). Hence, $\mathcal{R}_{Time}$ does not assume a particular $\widehat{\mathbf{Time}}$ implementation and leaves the actual definition open:

$$\mathcal{R}_{Time} : \mathbf{Time} \times \widehat{\mathbf{Time}} \to \{true, false\}$$

For an example, assume that the analysis runs with the simple time abstraction $Time_0$ that abstracts all concrete points in time to a single abstract time. $\mathcal{R}_{Time}$ for this time abstraction is defined by:

$$t \ \mathcal{R}_{Time_0} \ \widehat{t} = true$$

In general, the time abstraction must ensure that the following two conditions hold:

$$\begin{aligned} t \ \mathcal{R}_{Time} \ \widehat{t} \quad & \Rightarrow tick(t) \ \mathcal{R}_{Time} \ \widehat{tick}(\widehat{t}) \\ t \ \mathcal{R}_{Time} \ \widehat{t} \wedge \widehat{t} \sqsubseteq \widehat{t'} \quad & \Rightarrow t \ \mathcal{R}_{Time} \ \widehat{t'} \end{aligned}$$

All correctness relations implicitly obey the following conditions. Assume a concrete lattice L and its abstract counterpart $\widehat{L}$:

$$\begin{array}{ll} \bot_L \ \mathcal{R}_L \ \bot_{\widehat{L}} & = \mathit{true} \\ l \ \mathcal{R}_L \ \top_{\widehat{L}} & = \mathit{true} \\ \text{where } l \in L, \ \widehat{l} \in \widehat{L} & \end{array}$$

That is, the top value $\top$ represents all possible concrete values.

The correctness relation $\mathcal{R}$ now facilitates the formulation of the semantic correctness:

Theorem 1 (Semantic correctness). *For $\varsigma \in \mathcal{V}(pr)$ there exists $\widehat{\varsigma} \in \widehat{\mathcal{V}}(pr)$ such that $\varsigma \ \mathcal{R} \ \widehat{\varsigma}$.*

Proof. The proof works by induction over the state transitions and is included in Appendix B. □

4.4 Language-specific abstractions

To complete the analysis specification this section presents the abstract value domains and primop state transitions for analyzing Scheme and PreScheme programs.

The interface between the language independent core of the flow analysis and the language-specific part are the following semantic domains:

- *Basic values* such as integers, characters, and boolean values belong to $\widehat{\mathbf{Bas}}_{Lang}$.
- *Compound values* such as records, vectors, and arrays belong to $\widehat{\mathbf{Comp}}_{Lang}$.

Together with procedure values, these three domains form the sum domain for denotable values. The core semantics, however, does not need to have knowledge on values from $\widehat{\mathbf{Bas}}_{Lang}$ and $\widehat{\mathbf{Comp}}_{Lang}$: The semantics merely defines how these values flow as call arguments through the program. Hence, it is easy to parameterize the analysis for different source languages converted to the intermediate language by running it with suitable definitions of $\widehat{\mathbf{Bas}}_{Lang}$ and $\widehat{\mathbf{Comp}}_{Lang}$.

Literals and return values of primops are the source for basic and compound values in a program. The core semantics delegates the evaluation of literals and primops to the auxiliary functions $\widehat{\mathcal{K}}_{Lang}$ and $\widehat{\mathcal{P}}_{Lang}$:

- $\widehat{\mathcal{K}}_{Lang}$ *lit* is a function that evaluates the literal *lit* and returns a suitable abstract representative.
- $\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ prim\ \langle a_1, \ldots, a_n \rangle$ evaluates a call to the primitive operations *prim* with the given arguments.

Note that $\widehat{\mathcal{P}}_{Lang}$ is responsible for evaluating the primop call in the context of a regular call (see $\text{Prim}\widehat{\text{CallEval}}$ in Figure 4.2) as well as in the context of trivial calls (see Figure 4.5).

Hence, the interface for defining the language-specific parts of a flow analysis consists of four parts: The definition of domains for basic and compound values, and abstract evaluation functions for literals and primops. The following sections present two instances of this interface.

4.4.1 Abstracting PreScheme values

PreScheme [*Kelsey*, 1997] offers a rich set of distinct basic value types and two compound value types. This section introduces abstractions for analyzing PreScheme programs.

Basic values PreScheme knows the following basic value types: integer, float, boolean, character, string, a null type which has no values at all, an unit type with exactly one value, an "undefined" value type, two distinct types for input and output ports, the "external" type for values imported from a C program and the address type that represents memory addresses. Recall that PreScheme is a Scheme dialect that was designed to translate to C easily.

While it would be perfectly legal to abstract all types of concrete basic values to a single sort of abstract values, the result would be unnecessarily imprecise. A more precise abstraction distinguishes the values by their type. Thus, $\widehat{\mathbf{Bas}}_{Lang}$ is a sum lattice and each summand domain stands for a basic value type. For PreScheme the definition of $\widehat{\mathbf{Bas}}_{Lang}$ reads as follows:

$$\widehat{\mathbf{Bas}}_{Lang} = \widehat{\mathbf{Integer}} + \widehat{\mathbf{Address}} + \widehat{\mathbf{String}} + \ldots$$

The implementation described in Chapter 6 deals with all value types of PreScheme. In this section I will focus on the domains for integers and addresses as these domains demonstrate the abstraction techniques also used to model the remaining domains.

Consider integer numbers first. Evaluating an integer literal directly yields its value. That is, the precise value is available to the flow analysis — it is safe to assume that the abstract representative for this number has exactly one concrete counterpart. However, primop applications may also return integer values. Here, determining the precise value is not possible in general: For example, a primop may return an integer value that encodes the status of some I/O operation or the length of a file. The simplest solution involves representing all concrete integer values by one abstract representative. However, this solution is unnecessarily imprecise — whenever possible, the analysis should preserve the precise information.

A more suitable abstraction for integer values distinguishes between values known to be precise and a representative for imprecise values. The semantic domain for abstract integers reads as follows:

$$\begin{array}{lcl} \mathtt{Lit}_{Int} & = & \mathbb{N} \\ \widehat{\mathbf{Integer}} & = & \mathtt{Lit}_{Int} \cup \{\bot_{Int}, \top_{Int}\} \end{array}$$

$\widehat{\mathbf{Integer}}$ uses the literal values to represent precise abstract integers. The least element $\bot_{Int}$ and the greatest element $\top_{Int}$ represent imprecise integer values. Now, distinguishing whether an integer $\widehat{i} \in \widehat{\mathbf{Integer}}$ is precise or imprecise is straightforward:

$$\begin{array}{lll} \mathit{precise}(\widehat{i}) & \text{iff} & \widehat{i} \in \mathtt{Lit}_{Int} \\ \mathit{imprecise}(\widehat{i}) & \text{iff} & \widehat{i} \in \{\bot_{Int}, \top_{Int}\} \end{array}$$

The approximation relation $\sqsubseteq$ for abstract integers takes $\bot_{Int}$ and $\top_{Int}$ into account:

$$\widehat{i} \sqsubseteq \widehat{j} = \begin{cases} \widehat{i} = \widehat{j} & \widehat{i}, \widehat{j} \in \mathtt{Lit}_{Int} \\ \mathit{true} & \widehat{i} = \bot_{Int} \\ \mathit{true} & \widehat{j} = \top_{Int} \end{cases}$$

Each precise integer only represents itself and $\top_{Int}$ represents all abstract integers. Recall that $\widehat{\mathbf{D}}$ is a power lattice and thus abstract values are sets. Considered together with the definition of $\sqsubseteq$ on power lattices (see Section 4.1), the abstraction for integers now has the desired properties. For example, the following inequality holds because each element in the set on the left-hand side has a counterpart in the set on the right-hand side that represents it:

$$\{\mathsf{23}\} \sqsubseteq \{\mathsf{23}, \mathsf{42}\}$$

Thus, the abstract denotable value on the left-hand side represents the value on the right-hand side. $\top_{Int}$, however, represents all abstract integer values. Thus, the imprecise integer represents all precise integers:

$$\{\mathsf{23}, \mathsf{42}\} \sqsubseteq \{\top_{Int}\}$$

$\widehat{\mathcal{K}}_{Lang}$ yields a precise abstract integer while $\widehat{\mathcal{P}}_{Lang}$ always abstracts to $\top_{Int}$. That is, an integer literal value flows through the program as a precise values until it reaches a primop — then the precision degrades. This principle — distinguish precise and imprecise values — is also used for the domains that model characters, floats, and boolean values.

PreScheme offers functions for manipulating the memory on a low level. The type system distinguishes addresses from integer numbers using a separate type for addresses. From the perspective of the analysis, addresses are similar to integer values. However, preserving the exact value is less useful in this case. Thus, the analysis collapses all concrete memory addresses down to a single abstract address:

$$\widehat{\mathbf{Address}} = \{\widehat{addr}\}$$

Figure 4.8 summarizes the abstract semantic domains for PreScheme basic values: The domains listed build the summands for $\widehat{\mathbf{Bas}}_{Lang}$.

Compound values Compound values combine multiple distinct values into a single compound value. PreScheme offers two sorts of compound values: Records and vectors. Hence, the semantic domain $\widehat{\mathbf{Comp}}_{Lang}$ decomposes into two summands allowing the analysis to distinguish records and vectors:

$$\widehat{\mathbf{Comp}}_{Lang} = \widehat{\mathbf{Vector}} + \widehat{\mathbf{Record}}$$

Vectors Vectors in PreScheme have a fixed length and the static type system ensures that all elements of the vector are of the same type. During a program run, the length and type remain fixed. The programmer references the entries of a vector by integer indices. For the flow analysis, vectors pose a problem: As

$$
\begin{array}{lcl}
\texttt{Lit}_{Bool} & = & \text{all boolean literals} \\
\texttt{Lit}_{Char} & = & \text{all char literals} \\
\texttt{Lit}_{Float} & = & \text{all float literals} \\
\texttt{Lit}_{Int} & = & \text{all integer literals} \\
\texttt{Lit}_{Sym} & = & \text{all symbol literals} \\
\\
\widehat{\mathbf{Address}} & = & \{\widehat{addr}\} \\
\widehat{\mathbf{Boolean}} & = & \texttt{Lit}_{Bool} \cup \{\bot_{Bool}, \top_{Bool}\} \\
\widehat{\mathbf{Char}} & = & \texttt{Lit}_{Char} \cup \{\bot_{Char}, \top_{Char}\} \\
\widehat{\mathbf{Float}} & = & \texttt{Lit}_{Float} \cup \{\bot_{Float}, \top_{Float}\} \\
\widehat{\mathbf{InPort}} & = & \{\widehat{inport}\} \\
\widehat{\mathbf{Integer}} & = & \texttt{Lit}_{Int} \cup \{\bot_{Int}, \top_{Int}\} \\
\widehat{\mathbf{Null}} & = & \{\widehat{null}\} \\
\widehat{\mathbf{OutPort}} & = & \{\widehat{outport}\} \\
\widehat{\mathbf{String}} & = & \{\widehat{string}\} \\
\widehat{\mathbf{Undefined}} & = & \{\widehat{undef}\} \\
\widehat{\mathbf{Unit}} & = & \{\widehat{unit}\}
\end{array}
$$

Figure 4.8: Abstractions for PreScheme basic values

discussed before, integer values may not always be known precisely. That is, tracking the accesses to a certain field of a vector becomes impossible in general. Therefore, a conservative analysis has only one option to model vectors: Merge the values of all vector entries into a single abstract denotable value.

A further problem with vectors arises from the fact that vectors may contain arbitrary denotable values. That is, a naive model for vectors leads to a recursive domain for $\widehat{\mathbf{D}}$:

$$\widehat{\mathbf{Vector}} = \widehat{\mathbf{Integer}} \times \widehat{\mathbf{D}}$$

Here, the integer summand holds the vector length and the abstract denotable value is the union of all fields — introducing a recursion. To avoid a recursive definition a closer look at the implementation of vectors in the concrete semantics is helpful: A vector is a continuous memory region stored in the heap. That is, the value of a vector in the program is really a reference to a block of locations in memory. This, however, is easy to model in the flow analysis using the store. Consider this definition for abstract vectors:

$$\widehat{\mathbf{Vector}} = \widehat{\mathbf{Integer}} \times \widehat{\mathbf{Ref}}$$

This definition does not introduce a recursion into the domain equation for $\widehat{\mathbf{D}}$. Now, vectors are tuples consisting of an integer denoting the vector length and a reference. The reference points to a value in the store denoting the vector elements — collapsed into a single value.

The primop `make-vector` allocates a new vector and initializes it with the `null` value. Figure 4.9 shows the abstract transition rule for `make-vector`. The argument l indicates the length of the vector and i serves as a type sample: All entries of the new vector have the same type as i. The initial value for the

$$\widehat{\varsigma} = (\texttt{(make-vector}\ \langle c, l, i\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\{\widehat{ref}_v\}\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}$$

$$\text{where} \begin{cases} \widehat{t}' &= \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} &\in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, c \\ \widehat{l} &= \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, l \\ \widehat{ref}_v &= ((\tau, \widehat{t}'), \texttt{vector}) \\ \widehat{ref}_e &= ((\tau, \widehat{t}'), \texttt{elem}) \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup \left[\widehat{ref}_v \mapsto \{(\widehat{l}, \widehat{ref}_e)\},\ \widehat{ref}_e \mapsto \{\widehat{null}\}\right] \end{cases}$$

$$\widehat{\varsigma} = (\texttt{(vector-set!}\ \langle c, v, i, n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}$$

$$\text{where} \begin{cases} \widehat{t}' &= \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} &\in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, c \\ \widehat{vals} &= \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, v \\ \widehat{new} &= \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, n \\ \widehat{vref} &= \{\widehat{u} :\ \widehat{u} \in \widehat{vals}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\widehat{j}, \widehat{ref}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup \bigsqcup_{((\tau_i, \widehat{t}_i), s) \in \widehat{vref}} [((\tau_i, \widehat{t}_i), \texttt{elem}) \mapsto \widehat{new}] \end{cases}$$

Figure 4.9: PreScheme vector operations

elements of a vector is always `null`, however. The transition rules first advances the time to $\widehat{t}'$, evaluates the continuation argument c, and the argument l. Then, the rule creates two references for the location $(\tau, \widehat{t}')$: $\widehat{ref}_v$ uses the selector `vector` and is the reference to the value that represents the vector as a whole. $\widehat{ref}_e$ uses the selector `elem` and identifies the value that represents the vector elements. Finally, the transition rule adds both references to the store and produces an *apply* state for the continuation procedures using the reference $\widehat{ref}_v$ as the argument.

There are two primops that deal with vectors. `Vector-ref` returns the element stored at a given index of a vector, and `vector-set!` overwrites a value stored in a vector. `Vector-ref` only returns a value and has no side effect, and therefore is only used as a primop for trivial calls. Thus, the evaluation function for primops $\widehat{\mathcal{P}}_{Lang}$ has a case for `vector-ref`:

$$\widehat{\mathcal{P}}_{Lang}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}\, \texttt{vector-ref}\langle v, i\rangle = \bigsqcup_{((\tau, \widehat{t}'), s) \in \widehat{vref}} \widehat{\sigma}(((\tau, \widehat{t}'), \texttt{elem}))$$
$$\text{where } \widehat{v} = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}\, v$$
$$\widehat{vref} = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\widehat{j}, \widehat{ref}) \in \widehat{\sigma}(\widehat{u})\}$$

The argument v for `vector-ref` denotes the vector. The denotable value $\widehat{v}$ contains the abstract representatives for this vector. The set $\widehat{vref}$ is the subset of $\widehat{v}$ that contains all references to vector values. These references have `vector` as their selector since these are references to the vector as a whole. Exchanging the selector of these references with `elem` yields references that point to the

elements of the vector. Finally, the rule joins the abstract values retrieved from the store into one abstract value and returns this value.

Note that the rules ignores the index: The abstraction merges all element values into a single abstract value. Consequently the analysis looses precision when the program stores values in a vector. The analysis does not confuse vectors created at different call sites, though. Locations for heap values consist of a label and a point in time: The label identifies the constructor call and the time allows to distinguish dynamic instances of the constructor call. That is, every constructor call has its own store entry and, if there are multiple dynamic instances of this constructor call, each distinguishable call has its own entry.

Mutation of a vector works similarly but is more complex since this involves adding a transition rule. Figure 4.9 shows the transition rule for `vector-set!`. This rule evaluates the continuation argument c, the vector argument v, and the new value for the element n. Among the values in $\widehat{v}$ only references to vectors are to be considered: $\widehat{vref}$ contains these references. The rule then updates the store for each of this references replacing the selectors by `elem`.

Records A PreScheme program may define an arbitrary number of record types. The compiler translates these records directly to C structs. A record type definition introduces a new distinct record type with a fixed number of fields accompanied by selector and mutator functions that access or update a specific field of the record. Here is the record definition for a record which stores two-dimensional Cartesian coordinates as an example:

```
(define-record-type point :point
  (make-point x y)
  (x integer point-x set-point-x!)
  (y integer point-y set-point-y!))
```

The record has two fields, both of type integer, accessible with the selector procedures `point-x` and `point-y`. `Set-point-x!` and `set-point-y!` overwrite the values in a record's `x` and `y` fields with a new value. The procedure `make-point` is the record constructor that, given initial values for the fields, returns a new instance of the record.

The compiler maintains a table of all record types and the relevant information about the record fields, such as name and type. Each record type has an unique name — a literal symbol. Unlike in full Scheme, symbols are not denotable values in PreScheme. The compiler introduces symbols into intermediate language programs as named constants (see below for an example). To the analysis, knowing the fields of a record is important when creating fresh instances of records. The function *rfields* maps a record type name to the set of field names for this record type:

$$rfields : \texttt{Lit}_{Sym} \rightarrow \mathcal{P}(\texttt{Lit}_{Sym})$$

For the `point` record defined above, *rfields* returns the following:

$$rfields(\texttt{:point}) = \{\texttt{x}, \texttt{y}\}$$

There are three primops that deal with records:

```
(make-record type)
(record-ref rec type field)
(record-set! rec value type field)
```

The arguments named `type` and `field` are symbols and the argument named `rec` is an instance of a record. `Make-record` must not be confused with the constructor of a specific record type. It allocates heap space for a fresh instance of a record with the given type and initializes the fields with `null`. The record constructor is a generated procedure that calls `make-record` to allocate memory and `record-set!` to initialize the fields. The selectors and mutators are generated procedures that use `record-ref` to access a record field and `record-set!` to update a field, respectively. For both primops, the argument `field` specifies the field to access or update.

Note that the type and field arguments to the record primops are always symbols introduced by the compiler when generating the constructor, selector, and mutator procedures. That is, the analysis knows the values for this arguments precisely and this improves the precision of the record abstraction. The improved precision arises from two sources: First, the field name is always known. That is, the analysis may keep the abstract values for the fields separately and thereby avoid to confuse these values with values from other fields. Second, the exact type information allows distinguishing instances of records by their type. Consequently, the analysis does not merge values from one record of type A with values from another record of type B. Considered together, this means that the analysis merges values from the same field of possibly distinct concrete record instances, which are of the same type, but never confuses these values with the value for other fields or record types.

The abstraction for a record is the product lattice $\widehat{\mathbf{Record}}$. That is, an abstract record is a tuple consisting of a literal denoting the record type and a mapping from field names to references:

$$\widehat{\mathbf{Record}} = \mathtt{Lit}_{Sym} \times (\mathtt{Lit}_{Sym} \rightarrow \widehat{\mathbf{Ref}})$$

Like the definition of $\widehat{\mathbf{Vector}}$, this definition avoids recursion on $\widehat{\mathbf{D}}$.

Creating a fresh record using `make-record` adds entries for the record as a whole and the fields to the store — which is a side-effect. Hence, `make-record` may not be used in a trivial call and there is an transition rule for `make-record`. Figure 4.10 shows the transition rule. Consider the parts of the transition rule that extend the store. The rule extends the store with a reference to the record as a whole: $\widehat{ref}$. For each field that belongs to the record the rule adds a reference using the field name as the selector. Initially, the field references the least element in $\widehat{\mathbf{D}}$ — the empty set. In PreScheme `make-vector` calls `malloc` and therefore the initial value of the record fields is unspecified. The record implementation, however, always generates constructors that call `record-set!` immediately after `malloc` and initialize the fields. That is, PreScheme programs are not able to observe the fact that the fields are uninitialized right after being created. Hence, it is safe to model the initial value with $\perp_{\widehat{\mathbf{D}}}$. The value returned by `make-record` consists of $\widehat{null}$ and a reference since, as stated above, allocating memory for the record may fail.

Accessing a field of a record goes back to finding the reference for this field in the store. Here is the case of $\widehat{\mathcal{P}}_{Lang}$ for the primop `record-ref`:

$$\overline{\widehat{\varsigma} = (\texttt{(make-record}\ \langle c, t_{Sym}\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{res}\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{res} & = \{\widehat{ref}, \widehat{null}\} \\ \widehat{ref} & = ((\tau, \widehat{t}'), \texttt{record}) \\ \widehat{f} & = \bigsqcup_{fn \in rfields(t_{Sym})} [fn \mapsto ((\tau, \widehat{t}'), fn)] \\ \widehat{\sigma}' & = \widehat{\sigma} \sqcup [\widehat{ref} \mapsto \{(t_{Sym}, \widehat{f})\}] \sqcup \bigsqcup_{\widehat{r} \in \mathrm{rng}(\widehat{f})} [\widehat{r} \mapsto \bot_{\widehat{\mathbf{D}}}] \end{cases}$$

$$\overline{\widehat{\varsigma} = (\texttt{(record-set!}\ \langle c, r, t_{Sym}, f_{Sym}, n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{r} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,r \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,n \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{r}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (t_{Sym}, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma}' & = \widehat{\sigma} \sqcup \bigsqcup_{((\tau_i, \widehat{t}_i), s) \in \widehat{ref}} [((\tau_i, \widehat{t}_i), f_{Sym}) \mapsto \widehat{new}] \end{cases}$$

Figure 4.10: PreScheme record operations

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ \texttt{record-ref}\langle r, t_{Sym}, f_{Sym}\rangle = \bigsqcup_{((\tau, \widehat{t}'), s) \in \widehat{ref}} \widehat{\sigma}(((\tau, \widehat{t}'), f_{Sym}))$$

$$\begin{aligned} \text{where}\ \widehat{r} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,r \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{r}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (t_{Sym}, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \end{aligned}$$

The argument r evaluates to an abstract value that includes references to records. The record type expected by this `record-ref` is given as the argument t_{Sym} and is always a symbol. Therefore, the analysis knows the expected type precisely and filters $\widehat{r}$ for references to records of the type t_{Sym} — the result is the set $\widehat{ref}$. Replacing the `record` selector in the references of $\widehat{ref}$ with the field name in question (given as the symbol f_{Sym}) yields the set of references to the field values. Joining the values from the store for these references yields the return value.

Updating a field requires a transition rule (see Figure 4.10). First, the argument r evaluates to an abstract value which includes abstract references to records — these are the references to be considered. Using the exact information about the expected type given by the argument t_{Sym} the analysis places an additional restriction on the references $\widehat{ref}$: Only references to records with type t_{Sym} are considered. Updating the store works by joining the store with functions from the references (with the selector changed to the field name f_{Sym}) to the new value $\widehat{new}$.

Primops Altogether, PreScheme offers about 80 primitive operations all modeled in the flow analysis. The largest group (about 70 primops) among these primops are very simple primops. The `fl+` primop is an example for this class of primops. `Fl+` adds two floating point numbers and returns the result:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ \texttt{fl+}\langle e_0, e_1\rangle = \begin{cases} \{f_1 + f_2\} & e_i = \{f_i\} \ \wedge\ f_i \in \texttt{Lit}_{Float} \\ \top_{Float} & \text{otherwise} \end{cases}$$

Appendix C lists all PreScheme primops supported by the flow analysis.

Multiple return values Like Scheme, PreScheme allows the programmer to define procedures that return multiple values. Also, some primops return multiple values. `Open-input-file` is an example: Given the name of the file to open, the primop returns an input port and an integer indicating the status of the I/O operation just carried out. Calling such procedures and primops works as in regular Scheme: The special syntactic forms `call-with-values` [*Kelsey et al.*, 1998] and `receive` [*Stone*, 1999] split the return values and bind them to variables separately.

The PreScheme back end eliminates all calls to procedures or primops with multiple return variables from the program by expanding the parameter list of the continuation with extra parameters for the supplementary values returned. This technique only works if the continuation of the call is a lambda node. Hence, the front end enforces this restriction and rejects all programs that do not obey this condition. The static type system features *tuple types* to represent the type of a procedure with multiple return values.

This makes support for a distinct value type for multiple return values superfluous: All continuations have the matching numbers of parameters. To the flow analysis, however, multiple return values pose a problem: $\widehat{\mathcal{P}}_{Lang}$ is designed to return a single value. Thus, defining abstractions for these primops requires a different approach: Each primop with multiple return values demands its own transition rule. The following rule specifies the `open-input-file` primop:

$$\frac{prim_l = \texttt{open-input-file}}{\widehat{\varsigma} = ((prim_l\ \langle c, a\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{inport}\}, \{\top_{Int}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t'}\ c \end{cases}$$

The rule is a specialization of Prim$\widehat{\text{CallEval}}$ (see Figure 4.2) and transitions from an *eval* states to *apply* states corresponding to the invocation of continuations $\widehat{d}$. Besides `open-input-file`, there are a few more primops requiring similar rules. Appendix C.3 contains a complete list of these primops.

4.4.2 Abstracting Scheme values

The flow analysis also handles ordinary Scheme code. As stated earlier, the core of the analysis is only loosely coupled to the denotable values used in the programs — that is, the analysis does not need to know the structure of the domain $\widehat{\mathbf{D}}$ to analyze a program using values from $\widehat{\mathbf{D}}$. Hence, the literal

$$
\begin{array}{lcl}
\texttt{Lit}_{Bool} & ::= & \text{all boolean literals} \\
\texttt{Lit}_{Char} & ::= & \text{all char literals} \\
\texttt{Lit}_{Sym} & ::= & \text{all symbol literals} \\
\texttt{Lit}_{Int} & ::= & \text{all integer number literals} \\
\texttt{Lit}_{Rat} & ::= & \text{all rational number literals} \\
\texttt{Lit}_{Real} & ::= & \text{all real number literals} \\
\texttt{Lit}_{Complex} & ::= & \text{all complex number literals}
\end{array}
$$

$$
\begin{array}{lcl}
\widehat{\mathbf{Boolean}} & = & \texttt{Lit}_{Bool} \cup \{\bot_{Bool}, \top_{Bool}\} \\
\widehat{\mathbf{Char}} & = & \texttt{Lit}_{Char} \cup \{\bot_{Char}, \top_{Char}\} \\
\widehat{\mathbf{EmptyList}} & = & \{\widehat{empty}\} \\
\widehat{\mathbf{Eof}} & = & \{\widehat{eof}\} \\
\widehat{\mathbf{InPort}} & = & \{\widehat{inport}\} \\
\widehat{\mathbf{OutPort}} & = & \{\widehat{outport}\} \\
\widehat{\mathbf{RecType}} & = & \texttt{Lit}_{Int} \\
\widehat{\mathbf{String}} & = & \{\widehat{string}\} \\
\widehat{\mathbf{Symbol}} & = & \texttt{Lit}_{Sym} \cup \{\bot_{Symbol}, \top_{Symbol}\} \\
\widehat{\mathbf{Undefined}} & = & \{\widehat{undef}\} \\
\widehat{\mathbf{Unspecific}} & = & \{\widehat{unspecific}\}
\end{array}
$$

$$
\begin{array}{lcl}
\widehat{\mathbf{NumType}} & = & \{\texttt{integer}, \texttt{rational}, \texttt{real}, \texttt{complex}, \bot_{NumType}, \top_{NumType}\} \\
\widehat{\mathbf{Exactness}} & = & \{\texttt{exact}, \texttt{inexact}, \bot_{Exact}, \top_{Exact}\} \\
\widehat{\mathbf{NumValue}} & = & \texttt{Lit}_{Int} + \texttt{Lit}_{Rat} + \texttt{Lit}_{Real} + \texttt{Lit}_{Complex} \\
\widehat{\mathbf{Number}} & = & \widehat{\mathbf{NumType}} \times \widehat{\mathbf{Exactness}} \times \widehat{\mathbf{NumValue}}
\end{array}
$$

Figure 4.11: Abstractions for Scheme basic values

abstraction function $\widehat{\mathcal{A}}$ and the primop evaluation function $\widehat{\mathcal{P}}_{Lang}$ need to be re-defined to prepare the analysis for Scheme.

In this this Section, I define a value domain $\widehat{\mathbf{D}}$ suitable for deriving a precise analysis of Scheme programs and discuss the Scheme abstraction functions and some of the additional state transitions for complex Scheme primops. A complete listing of the primop semantics is included in Appendix D.

Basic values Figure 4.11 shows the summands of the domain $\widehat{\mathbf{Bas}}_{Lang}$: For each distinct basic value type there is a corresponding domain. The domains for booleans, characters, ports, strings and the undefined value work exactly as their PreScheme counterparts and use the same abstraction techniques.

Input operations such as `read-char` either return a character read from a port or the end of file value [*Kelsey et al.*, 1998]. Scheme has a distinct value that signals the end of a file. There is only one value of this type in Scheme 48 and consequently the abstract domain for this value type $\widehat{\mathbf{Eof}}$ is a singleton set.

The abstract values from $\widehat{\mathbf{Undefined}}$ are not denotable values in Scheme. The special *undefined* value exists in the Scheme 48 to reflect bound but unas-

signed global variables. The flow analysis must be able to simulate the situation just described. Thus, *undefined* has the abstract counterpart $\widehat{undef}$ which flows through the program like other basic values.

Primitive operations with side effects usually do not return a value. For technical reasons, however, every primop application in the Scheme implementation has to return a value. If there is no meaningful value to return, primops return the *unspecific* value. In contrast to *undefined*, *unspecific* is denotable (if useless). The following interaction in the Scheme 48 REPL serves as an example:

```
> (if #f 42)
#{Unspecific}
```

The abstract counterpart for the *unspecific* value is $\widehat{unspecific}$. The $\widehat{\textbf{Unspecific}}$ domain contains just this value.

Note that the empty list has a distinct type. The analysis uses the domain $\widehat{\textbf{EmptyList}}$ for the empty list.

Number values Scheme arranges the numerical types in a so-called *numeric tower* in which each numerical type is a subset of the type above [*Kelsey et al.*, 1998]. The types are: *integer*, *rational*, *real*, and *complex*. Integer are the smallest type and each integer number is also a rational number, a real number, and a complex number. The primitive operations on numbers work for all subtypes and the type of the return value depends on the argument types.

For example, given an integer and a rational number, the `+` operation returns a rational number — the largest type of the arguments types. For optimization purposes, obtaining detailed information on the number types is useful: If the flow analysis proves that the arguments of a numeric primop at some point in the program only contains numerical values of a certain types then the compiler may replace the general version of the primitive operation, which contains a dispatch on the argument types, with a faster specific version that omits the check.

Besides the number type, numerical values in Scheme have one additional property that encodes whether the value is derived from an *exact* or *inexact* operation. A value arisen from the evaluation of a literal, for example, is always an exact value. The primop for division, however, does not return an exact result in all cases. Thus, the value returned from a division may be marked as being inexact. Note that the *exactness* of a number is orthogonal to its type.

That is, a Scheme number has three orthogonal characteristics: type, exactness, and value. The flow analysis models all of these characteristics and thus can often compute precise information on the number values used in a program. Elements from $\widehat{\textbf{Number}}$ represent concrete Scheme numbers and consist of type information (specified by $\widehat{\textbf{NumType}}$) exactness information given by $\widehat{\textbf{Exactness}}$, and a value from $\widehat{\textbf{NumValue}}$:

$$
\begin{aligned}
\widehat{\textbf{NumType}} &= \{\texttt{integer}, \texttt{rational}, \texttt{real}, \texttt{complex}, \bot_{NumType}, \top_{NumType}\} \\
\widehat{\textbf{Exactness}} &= \{\texttt{exact}, \texttt{inexact}, \bot_{Exact}, \top_{Exact}\} \\
\widehat{\textbf{NumValue}} &= \texttt{Lit}_{Int} + \texttt{Lit}_{Rat} + \texttt{Lit}_{Real} + \texttt{Lit}_{Complex} \\
\widehat{\textbf{Number}} &= \widehat{\textbf{NumType}} \times \widehat{\textbf{Exactness}} \times \widehat{\textbf{NumValue}}
\end{aligned}
$$

If $\widehat{\mathcal{A}}$ evaluates the literal node `42`, for example, the analysis simulates this with the value

$$(\texttt{integer}, \texttt{exact}, 42)$$

In this case type, value, and exactness are known precisely and the abstract representative reflects this. In contrast to the evaluation of literals, the abstract counterpart of an arithmetic operation such as `*` can not compute the precise value of the result in general. That is, such an operation returns the abstract integer that represents all possible integers: $\top_{NumVal}$. So, for a call to `*` with exact integer arguments the primop evaluation function $\widehat{\mathcal{P}}_{Lang}$ returns the following abstract value:

$$(\texttt{integer}, \texttt{exact}, \top_{NumVal})$$

That is, the analysis preserves the precise exactness and type information. In fact, the Scheme standard requires many arithmetic primops to preserve the exactness. Appendix D lists these Scheme primops.

This machinery, however, is not yet sufficient to model numbers correctly. For an example, consider the addition of numbers. Adding the rational number `1/2` to itself yields the result `1` — a number that may be represented as a rational or an integer. There is no simple rule to determine the number type of the result value. Consequently, the analysis can not compute the type of the return value, because this requires exact knowledge about the argument values — which may not be available. To cope with this problem, the analysis computes an abstract value which contains multiple abstract numbers. For the addition of `1/2` mentioned above, the following abstract value represents the return value:

$$\{(\texttt{integer}, \texttt{exact}, \top_{NumVal}), (\texttt{rational}, \texttt{exact}, \top_{NumVal})\}$$

That is, there a representative for all integer and all rational numbers. Clearly, $(\top_{NumType}, \texttt{exact}, \top_{NumType})$ would also be a valid abstraction. However, $\top_{NumType}$ is a unnecessary imprecise abstraction since this discards the knowledge that this abstract value will not contain a number of type real or complex.

Computing the correct type and exactness characteristics of a number when applying an arithmetic primop requires a few auxiliary functions. Consider the auxiliary functions for the number types first.

The type of the result values for almost all arithmetic operations depends on the types of the arguments. The function *ntypes* computes the set of number types found in a denotable value or in an argument vector:

$$\begin{array}{lcl} ntypes(\widehat{d}) & = & \bigsqcup\{nt : (nt, e, nv) \in \widehat{d}\} \\ ntypes(\langle \widehat{d}_0, \ldots, \widehat{d}_n \rangle) & = & \bigsqcup_{0 \leq i \leq n} ntypes(\widehat{d}_i) \end{array}$$

As stated before, the relationship between the types of the argument values and the result value type is not simple. So the return value type is not always the greatest type found in the argument values — in fact any smaller type must also be considered. The relation $\prec$ orders the representatives for types from $\widehat{\textbf{NumType}}$ by their size:

$$\bot_{NumType} \prec \texttt{integer} \prec \texttt{rational} \prec \texttt{real} \prec \texttt{complex} \prec \top_{NumType}$$

The $\prec$ relation facilitates defining a function *cntypes* that returns the set of all type that are smaller than a given type:

$$\begin{array}{lcll} cntypes(nt) & = & \{nt' \in \widehat{\mathbf{NumType}} \mid nt' \prec nt\} & nt \in \widehat{\mathbf{NumType}} \\ cntypes(NT) & = & \bigsqcup_{nt \in NT} cntypes(nt) & NT \in \mathcal{P}(\widehat{\mathbf{NumType}}) \end{array}$$

To determine the exactness of the resulting value, these primops use the auxiliary function *exact*?. Applied to an abstract value or an argument list *exact*? detects whether all abstract numbers — regardless of their type — are exact:

$$\begin{array}{lcl} exact?(\widehat{d}) & \Leftrightarrow & \forall\, (nt, e, nv) \in \widehat{d}:\ e = \texttt{exact} \\ exact?(\langle \widehat{d}_0, \ldots, \widehat{d}_n \rangle) & \Leftrightarrow & \forall\, 0 \leq i \leq n : exact?(\widehat{d}_i) \end{array}$$

Akin to *exact*?, *inexact*? determines whether an abstract value or a vector of abstract values contains only number values known to be inexact:

$$\begin{array}{lcl} inexact?(\widehat{d}) & \Leftrightarrow & \forall\, (nt, e, nv) \in \widehat{d}:\ e = \texttt{inexact} \\ inexact?(\langle \widehat{d}_0, \ldots, \widehat{d}_n \rangle) & \Leftrightarrow & \forall\, 0 \leq i \leq n : inexact?(\widehat{d}_i) \end{array}$$

With the machinery for exactness and number types in place, it is now possible to define the arithmetic primops. Here is the transition rule:

$$\frac{}{\widehat{\varsigma} = (\texttt{((+}\ \langle c, a_0, a_1 \rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle \widehat{r} \rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, c \\ \widehat{a}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}'\, a_i \\ \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{r} & = \{(nt, e, \top_{NumVal}) : nt \in cntypes(ntypes(\langle \widehat{a}_0, \widehat{a}_1 \rangle))\} \\ e & = \begin{cases} \texttt{exact} & exact?(\langle \widehat{a}_0, \widehat{a}_1 \rangle) \\ \texttt{inexact} & inexact?(\langle \widehat{a}_0, \widehat{a}_1 \rangle) \\ \top_{Exact} & \text{otherwise} \end{cases} \end{cases}$$

This transition rule first advances the time to $\widehat{t'}$ and evaluates the call arguments in the usual manner using $\widehat{\mathcal{A}}$. For `+` the exactness of the return values depends on the exactness of the argument values. The auxiliary functions *exact*? and *inexact*? first determine whether the evaluated arguments $\widehat{d}_i$ consist exclusively of exact or inexact values. If this is the case, it is safe to say that the resulting value is either exact or inexact. If the arguments have mixed exactnesses, however, the analysis is not able to detect the exactness and uses $\top_{Exact}$. Bear in mind, that even when used with a polyvariant set of abstract times, the analysis merges the argument values for multiple dynamic instances of a call. For an example, consider the following scenario in a concrete program run: At some point in time the program calls the function in question with exact argument values, and at a later point in time the program calls the same function with inexact argument values. Clearly, the function will be either called with exact or inexact numbers, but never with numbers of mixed exactness. Hence, $\top_{Exact}$ is the correct abstraction for the exactness of the return value.

The return value $\widehat{r}$ consists of multiple abstract numbers, one for each number type. *ntypes* computes the set of all number types appearing in the argument

vector. *cntypes* then completes the set by adding all smaller number types. The result is a set of types which includes all types starting at `integer` up to the greatest number type found in the arguments.

The transition rule for `+` is a typical example for an arithmetic primop. Many other primops work in a similar way or only exhibit minor differences and therefore do not require further explanations. Appendix D lists a subset of the arithmetic primops supported by the flow analysis.

Compound values The Scheme 48 system knows the following compound values: cells, pairs, records, and vectors. Cells are a simple mechanism for indirection: The constructor `make-cell` creates a fresh cell filled with an initial value and returns the cell. `Cell-ref` takes a cell and extracts the value stored in the cell. `Cell-set!` updates the value in a given cell overwriting the old value. Thus, cells may also be considered as records or vectors with exactly one field. The Scheme 48 system uses cells to store the values of lexical variables that are subject to destructive updates. That is, if the program modifies the value of a variable using `set!` the system binds the variable to a fresh cell and replaces every variable reference to a call to `cell-ref` and every `set!` to a call to `cell-set!`.

All compound values in Scheme 48 have an uniform representation as so-called *stored objects* (or *stobs* for short) in the virtual machine. A stored object is a heap value with a header identifying the value as a stob. Stobs have a field that encodes the Scheme type this stob belongs to. The rest of the stob data is specific to the type: A stob representing a cell only contains one further value slot to hold the cell's value. Pair stobs include two value slots for the car and cdr values. Vector stobs contain a field for the length of the vector and space for the vector elements. Record stobs consist of a slot for the record type and a slot for each field.

For example, the code for creating a pair using the call `(cons 42 23)` translates to

```
(make-stob '0 '42 '23)
```

in the intermediate language. `Make-stob` is the primop that creates a fresh stob and returns it. The integer literal `0` encodes the stob type — zero stands for pairs. Further arguments are the initial values for the stob fields. Note that the integer encoding the stob type is always immediate and is fixed for the life-time of the stob. Thus, the analysis always exactly knows the type of the stob created by `make-stob`. The primitive operations `stob-ref` and `stob-set!` extract a single field from a stob or modify a field.

The flow analysis represents stored objects by values from $\widehat{\mathbf{Stob}}$, defined by the following domain equation:

$$\widehat{\mathbf{Stob}} = \mathtt{Lit}^*_{Lang} \times (\mathtt{Lit}_{Int} \rightarrow \widehat{\mathbf{Ref}})$$

That is, an abstract stob is a tuple consisting of a vector and a function. The vector contains literal values and stores meta information on the stob value such as the stob type. The vector always has at least one entry for the stob type. For better readability, I will write the name of the stob type in typewriter font instead of the type number. The remaining elements of the vector encode other information that is specific to the stob type: Records, for example, also contain

a record type and vectors contain length information. The second component of the tuple simulates the fields of the stob as a function. Given an index this function returns the abstract reference for the field value.

Consider the following example. The call `(make-stob '0 '23 '42)` at label τ and occurring at analysis time $\widehat{t}$ creates an abstract stob that encodes a pair. This stob has the following structure:

$$(\langle \texttt{pair} \rangle, \left[\texttt{0} \mapsto (\tau, \widehat{t}), \texttt{1} \mapsto (\tau, \widehat{t})\right])$$

Like the representation of PreScheme records and vectors this representation does not introduce a recursive domain. The values that belong to a stob are not part of the stob, the stob merely contains references to field values. Three entries in the store belong to the stob shown above:

$$\begin{aligned}
\widehat{\sigma}(((\tau, \widehat{t}), \texttt{stob})) &= \{(\langle \texttt{pair} \rangle, \left[\texttt{0} \mapsto (\tau, \widehat{t}), \texttt{1} \mapsto (\tau, \widehat{t})\right])\} \\
\widehat{\sigma}(((\tau, \widehat{t}), \texttt{0})) &= \{(\texttt{integer}, \texttt{exact}, 23)\} \\
\widehat{\sigma}(((\tau, \widehat{t}), \texttt{1})) &= \{(\texttt{integer}, \texttt{exact}, 42)\}
\end{aligned}$$

The first entry represents the stob itself as indicated by the selector `stob` and for each field there is a separate store entry. Note how the selector distinguishes values stored under the same abstract heap location. The representation of stobs includes a function that maps each field name to the reference in the store that contains the value of the field. Two reasons motivate this choice.

The function mapping field names to references makes the stob representation independent of the abstract function used for references in the analysis. Recall that references as discussed in this dissertation consist of a location and a selector. That is, given a location and the set of selectors for a stob the set of references for this stob is easy to construct. Depending on the purpose of the analysis, the user may want to run the analysis using a more (or less) precise store abstraction. In such an abstraction the correlation between the references to a stob and the references for the stob fields may not be obvious.

The second reason is technical. From an implementation perspective, knowing the values reachable in the store through a given stob turns out to be very useful: The implementation of the flow analysis uses a garbage collector to improve the speed and precision of the analysis (see Chapter 6). During the analysis of large programs, garbage collections occur often and are run-time intensive. That is, the run time of the garbage collector is critical to the complete run time. The novel implementation technique for such a garbage collector depicted in Chapter 6 borrows ideas from a marking garbage collector and directly uses the references to efficiently trace reachable values.

Figure 4.12 shows the three transitions rules for `make-stob`. The first transition rule applies when `make-stob` is used to create a pair or a cell, the second if the stob is a record and the last rule concerns vectors. The rules are very similar. All these rules advance the time, evaluate the continuation argument c, and evaluate the arguments a_i, the initial field values.

Consider the `make-stob` rule for vector stobs first. As in the PreScheme case, the analysis merges all values from all fields into a single abstract value. The reason is simple: For most accesses to vectors, the analysis cannot determine the exact index. Thus, it makes no sense to keep these values in separate fields in the first place. The argument s denotes the vector length. The abstract stob stores this information along with the stob type in the first component of stob

$$\frac{t_{Int} \notin \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = (\texttt{(make-stob}\ \langle c, t_{Int}, a_1, \ldots, a_n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' &= \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_i \\ \widehat{f} &= \bigsqcup_{1\le i\le n} \left[i \mapsto ((\tau, \widehat{t}'), i)\right] \\ \widehat{ref} &= ((\tau, \widehat{t}'), \texttt{stob}) \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup \left[((\tau, \widehat{t}'), i) \mapsto \widehat{v}_i\right] \sqcup \left[\widehat{ref} \mapsto \{(\langle t_{Int}\rangle, \widehat{f})\}\right] \end{cases}$$

$$\frac{t_{Int} = \texttt{record}}{\widehat{\varsigma} = (\texttt{(make-stob}\ \langle c, t_{Int}, rt_{Int}, , a_1, \ldots, a_n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' &= \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_i \\ \widehat{f} &= \bigsqcup_{1\le i\le n} \left[i \mapsto ((\tau, \widehat{t}'), i)\right] \\ \widehat{ref} &= ((\tau, \widehat{t}'), \texttt{stob}) \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup \left[((\tau, \widehat{t}'), i) \mapsto \widehat{v}_i\right] \sqcup \left[\widehat{ref} \mapsto \{(\langle \texttt{record}, rt_{Int}\rangle, \widehat{f})\}\right] \end{cases}$$

$$\frac{t_{Int} = \texttt{vector}}{\widehat{\varsigma} = (\texttt{(make-stob}\ \langle c, t_{Int}, s, a_1, \ldots, a_n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' &= \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v}_i &\in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_i \\ \widehat{f} &= [0 \mapsto ((\tau, \widehat{t}'), \texttt{elements})] \\ \widehat{ref} &= ((\tau, \widehat{t}'), \texttt{stob}) \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup \left[((\tau, \widehat{t}'), \texttt{elements}) \mapsto \bigsqcup_{1\le i\le n} \widehat{v}_i\right] \sqcup \left[\widehat{ref} \mapsto \{(\langle \texttt{vector}, s\rangle, \widehat{f})\}\right] \end{cases}$$

Figure 4.12: Transition rules for `make-stob`

tuple. The rule augments the store with an entry for the field using the selector `0` and also adds the entry that denotes the vector as a whole. The reference to this value is also the value used as the argument for the subsequent application of the continuation.

For cells and pairs (first rule of Figure 4.12) the situation is similar. However, this rule adds an entry for each field value. Like in the virtual machine implementation, the flow analysis uses an index to identify a field and never merges abstract values from distinct fields. It may, however, merge values from the same fields of distinct stob instances.

The second rule of Figure 4.12 concerns records. I will discuss this rule in the context of the other record-related primops later in this section.

The primop `stob-ref` extracts a given field of a stob. Figure 4.13 shows the transition rule for this primop. Note that `stob-ref` takes four arguments: The continuation of the call c, the stob value a_1, an integer literal i_{Int} that identifies the field to access, and the stob type expected t_{Int}. The virtual machine uses the stob-type argument to detect type errors. If a_1 evaluates to a stob value that does not match the type t_{Int} this raises a type error. For example, a call to `stob-ref` implementing `car` that is called with a record must result in a type error. Hence, `stob-ref` compares the type of the stob to access with the type passed as arguments and eventually raises an error.

In the abstract semantics this check does not make sense since the abstract values passed to `stob-ref` may contain false positives. The type information is, however, useful to the analysis. Like the field name, the stob type is always given in form of a literal and consequently the analysis knows the exact value. This allows the transition rule for `stob-ref` to filter all abstract stob values for stob values of the matching type and merge the values from the fields in question.

Besides `stob-ref` there is also `stob-indexed-ref` which is the basis for implementing Scheme procedures like `vector-ref` that take an index argument computed at run time. See Appendix D.2 for the transition rule of `stob-indexed-ref`.

Records A program may define an arbitrary number of distinct record types. The Scheme 48 virtual machine represents all these record types as stobs marked with the stob type `record`. This distinguishes records from other stobs, but does not yet distinguish between record types. To achieve this, the virtual machine uses a field in the stob data to store information on the record type. This field contains another record of a special type, a *record type record*. Note that in Scheme 48 there is more than one record library (e. g. SRFI 9 records [*Kelsey*, 1999] and the built-in records). All these record systems are implemented in terms of the fundamental record type used in the virtual machine — the stob marked with `record`. In this work, the term record type always refers to this fundamental record type unless declared the opposite.

Record-type records store information on a record type such as an unique type id, the number of fields, a type name, and a procedure to print records. Programs — except for low-level code such as the debugger — usually do not have access to the record-type records.

Consider the following example for records and record-type records. To define a new record type in Scheme, the programmer uses `define-record-type`:

$$\frac{t_{Int} \notin \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = ((\texttt{stob-ref}\ \langle c, a_1, t_{Int}, i_{Int}\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{r} & = \bigsqcup_{((\tau, \widehat{t''}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau, \widehat{t''}), i_{Int})) \end{cases}$$

$$\frac{t_{Int} \neq \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = ((\texttt{stob-set!}\ \langle c, a_1, t_{Int}, i_{Int}, a_2\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_2 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \left[((\tau', \widehat{t''}), i_{Int}) \mapsto \widehat{new}\right] \end{cases}$$

Figure 4.13: Accessing a stob field

```
(define-record-type pare :pare
  (kons kar kdr)
  pare?
  (kar kar set-kar!)
  (kdr kdr set-kdr!))
```

This definition creates a constructor, a type predicate, and selector and mutator functions for this record type. Also, it creates a new record-type record and binds it to `:pare`:

```
> :pare
#{Record-type 47 pare}
> ,inspect
#{Record-type 47 pare}

 [0: resumer] #t
 [1: uid] 47
 [2: name] 'pare
 [3: field-names] '(kar kdr)
 [4: number-of-fields] 2
 [5: discloser] '#{Procedure 1237 [...]}
```

During a pre-pass, the analysis finds all references to such record-type records and generates a list of all record-type records used in the program. Most of the

$$\frac{\mathcal{N} = \texttt{checked-record-ref}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,a_1 \\ \widehat{ref} & = \{\widehat{u} : \widehat{u} \in \widehat{v} \wedge \widehat{u} \in \widehat{\mathbf{Ref}} \wedge (\langle\texttt{record}, t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{r} & = \bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau', \widehat{t''}), i_{Int})) \end{cases}$$

$$\frac{\mathcal{N} = \texttt{checked-record-set!}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}, a_2\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,a_1 \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,a_2 \\ \widehat{ref} & = \{\widehat{u} : \widehat{u} \in \widehat{v} \wedge \widehat{u} \in \widehat{\mathbf{Ref}} \wedge (\langle\texttt{record}, t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \left[((\tau', \widehat{t''}), i_{Int}) \mapsto \widehat{new}\right] \end{cases}$$

Figure 4.14: Scheme record operations

information stored in a record-type record is irrelevant for the purpose of the analysis. Thus, the flow analysis represents record types simply by their unique type id (see the definition of $\widehat{\mathbf{RecType}}$ in Figure 4.8).

The virtual machine checks all accesses to record fields and raises a type error if the record if not of the expected record type. Thus, accessing the field of a record consists of two checks. First, the virtual machine checks whether the value in question is a stob that represents a record. If this is the case, the second check matches the record-type field against the expected record type. The primitive operation `checked-record-ref` returns the value of a record field, and the mutator primop `checked-record-set!` updates a field of a record.

Figure 4.14 shows the transition rules for the primitive record operations. Both rules are very similar to the rules for `stob-ref` and `stob-set!`, but also take the record type into account. Both rules evaluate a_1 — the record value — and restrict the abstract values to references to stob values that represent a record of the matching type t_{Int}: $\widehat{ref}$ contains these references. The transition rule for `checked-record-ref` joins the values stored under the references in $\widehat{ref}$ and produces an *apply* state that applies the continuation to the joined value. The transition rule for the mutation primop `checked-record-set!` joins the values stored under the references in $\widehat{ref}$ with the new value given as the argument a_2.

Procedures with variable arity Scheme procedures may have variable arity. For an example consider the following procedure definition:

```
(lambda (x y . more)
  (+ x y (apply + more)))
```

This procedure has two mandatory arguments, `x` and `y`, and the rest list argument `more`. The dot in the argument list indicates that the following argument is the rest list argument. The procedure must be called with at least two arguments. For a call with more than two arguments, the Scheme system allocates a fresh list containing the remaining arguments and binds this list to `more` for the call.

Adding support for the rest list argument to the flow analysis raises some technical problems. First, the intermediate language must be able to distinguish lambda terms with a fixed number of arguments from n-ary lambda terms. Second, the transition rule $\widehat{\text{ApplyClos}}$ (see Section 4.2) becomes considerably more complex since it needs to match the arguments against the parameters and eventually creates a rest list.

The CPS transformation only introduces lambda expressions with a fixed number of arguments. So, only user-defined procedures may be n-ary. The superscript n marks a lambda term as being n-ary:

$$(\lambda^{n\mathsf{P}}(v_1^u \dots v_n^u)\ call)_\tau$$

For lambda expressions marked this way, the last element of the argument list v_n^u is always the rest list argument. A procedure without mandatory argument consequently only has one argument for the rest list.

To allocate a fresh list for the remaining arguments of a call $\widehat{\text{ApplyClos}}$ uses the auxiliary function $\widehat{listify\text{-}args}$. Given a vector with n evaluated arguments, an index k into this vector, and an abstract state this function augments the store with a list containing the elements from k to n of the argument vector. $\widehat{listify\text{-}args}$ returns the updated store and the reference to the rest list:

$$\widehat{listify\text{-}args} : \widehat{\mathbf{D}}^* \times \mathbb{N} \times \widehat{\mathbf{Apply}} \to \widehat{\mathbf{D}} \times \widehat{\mathbf{Store}}$$

$$\widehat{listify\text{-}args}(\langle v_0, \dots, v_n\rangle, k, \widehat{\varsigma}) = (\{\widehat{stob_k}\},\ \widehat{\sigma}')$$

where

$$\begin{aligned}
\widehat{stob}_{n+1} &= \{\widehat{empty}\}\\
\widehat{stob}_i &= ((\tau(a_i), \widehat{t}),\ \texttt{stob})\\
\widehat{car}_i &= ((\tau(a_i), \widehat{t}),\ 0)\\
\widehat{cdr}_i &= ((\tau(a_i), \widehat{t}),\ 1)\\
\widehat{\sigma}' &= \widehat{\sigma} \sqcup \bigsqcup_{k \le i \le n} [\widehat{stob}_i \mapsto \{(\langle \texttt{pair}\rangle,\ [0 \mapsto \widehat{car}_i,\ 1 \mapsto \widehat{cdr}_i])\}]\\
&\quad \sqcup \bigsqcup_{k \le i \le n} [\widehat{car}_i \mapsto \widehat{v}_i]\\
&\quad \sqcup \bigsqcup_{k \le i \le n} [\widehat{cdr}_i \mapsto \widehat{stob}_{i+1}]
\end{aligned}$$

with

$$(\texttt{(}prim\ \langle a_0, \dots, a_n\rangle\texttt{)}, \widehat{\beta}'', \widehat{ve}'', \widehat{\sigma}'', \widehat{t}'') \mathrel{\widehat{\Rightarrow}} \widehat{\varsigma}$$

Note that $\widehat{listify\text{-}args}$ expects an *eval* state as its third argument — this state

is the predecessor of the *apply* state calling $\widehat{\textit{listify-args}}$. By construction every *apply* state except for the initial states has an *eval* state as its predecessor: $\widehat{\Rightarrow}$ never goes from an *apply* state to an *apply* state and $\widehat{\Rightarrow}$ creates a separate *apply* state for each closure an operator expression evaluates to (see Section 4.2). Considered together, all *apply* states which are not initial states must have exactly one predecessor which is an *eval* state.

$\widehat{\textit{listify-args}}$ uses the predecessor state to generate fresh store references. The labels of the call arguments $\tau(a_i)$ together with the time of the *apply* state provide the location. This choice ensures a high grade of precision: The label discriminates different call sites in the program and the abstract time may disambiguate dynamic instances of the same call.

The return value of $\widehat{\textit{listify-args}}$ is a tuple consisting of a reference to the first pair in the list $\widehat{stob_k}$ (or $\widehat{empty}$) and the store augmented with the new pairs.

Equipped with this auxiliary function, the version of $\widehat{\text{ApplyClos}}$ for n-ary procedures looks like this:

$$\frac{\text{length}(\hat{c}^* \S \hat{d}^*) \geq n - 1}{\hat{\varsigma} = (\{((\lambda^{n\mathsf{P}}(p_1 \ldots p_n)\ \ \mathit{call})_\tau, \hat{\beta}, \hat{t_b})\}, \hat{c}^*, \hat{d}^*, \hat{ve}, \hat{\sigma}, \hat{t}) \widehat{\Rightarrow} (\mathit{call}, \hat{\beta}', \widehat{ve'}, \widehat{\sigma'}, \hat{t})}$$

$$\text{where} \begin{cases} (\hat{r}, \widehat{\sigma'}) & = \widehat{\textit{listify-args}}((\hat{c}^* \S \hat{d}^*), n, \hat{\varsigma}) \\ \hat{\beta}' & = \hat{\beta}[p_i \mapsto \hat{t}] \\ \widehat{ve'} & = \hat{ve} \ \sqcup \bigsqcup\limits_{1 \leq j < n} [(p_j, \hat{t}) \mapsto (\hat{c}^* \S \hat{d}^*)_j] \ \sqcup \ [(p_n, \hat{t}) \mapsto \hat{r}] \end{cases}$$

This additional rule only applies to closures over n-ary lambdas. Technically, the original $\widehat{\text{ApplyClos}}$ must also be modified to exclude exactly those closures. The rule for n-ary procedures uses $\widehat{\textit{listify-args}}$ to wrap the arguments in the concatenated argument vector $\hat{c}^* \S \hat{d}^*$ that exceed the argument count n into the rest-list argument $\hat{r}$. Calling $\widehat{\textit{listify-args}}$ may result in an updated store $\widehat{\sigma'}$. Then, the rule proceeds as its counterpart $\widehat{\text{ApplyClos}}$: It updates the binding environment to reflect the new binding time for p_i and merges the new argument values — including the rest list — into the variable environment.

Complete programs As described in Section 2.4.2 the input to the analysis is not source code but rather a Scheme procedure. The byte-code optimizer converts this procedure to a single CPS node. However, the optimizer does not pay attention to the free variables of the procedure it is translating. That is, the flow analysis must first close the program over these variables. Therefore the analysis traverses the byte-code of the procedure in a pre-pass (using the Scheme 48 byte-code parser) and builds the transitive closure over all free variables. The result is a mapping from locations to Scheme values.

The fact that the program is given in form of Scheme values poses an additional problem: So far, there is only an abstraction function that creates abstract value on basis of literal nodes — that is, on basis of some source code. The source code, however, that led to the Scheme values referenced in the program in general, is not available. Here, a second abstraction function is necessary: The abstraction function $\widehat{\mathcal{S}}$ turns a concrete Scheme value into its abstract counterpart:

$$\widehat{\mathcal{S}} : \mathbf{D}_{Scheme} \rightarrow (\widehat{\mathbf{D}} + \mathtt{Lam})$$

The mode of operation for $\widehat{\mathcal{S}}$ is as follows: If the value is a procedure[1], then $\widehat{\mathcal{S}}$ uses the byte-code optimizer to convert the byte code of the procedure to a CPS representation. That is, $\widehat{\mathcal{S}}$ returns a lambda node. Converting other values, such as characters, symbols, booleans, integers, rational numbers, real numbers, complex numbers, and the EOF object yields an abstract value of the corresponding abstract value type.

For compound values referenced in the program the situation is more complicated: These values abstract to references into the store which holds the actual values. Here, the implementation differs from the analysis as presented so far. Abstract locations in the implementation really are a sum domain of calls, global variables, and heap locations in the Scheme heap. That is, the analysis injects the heap location used by the Scheme 48 directly into the domain of abstract locations. The store created by this pre-pass is the initial store of the analysis.

To abstract the fields of a compound value, the pre-pass calls $\widehat{\mathcal{S}}$ recursively. For a Scheme pair, for example, $\widehat{\mathcal{S}}$ returns a reference to a stob and augments the store with entries for the stob, car, and cdr. This technique also works for records: As described in the preceding paragraphs there is a record-type record for each record type. Among other things, the record-type record also stores the number of fields, which allows $\widehat{\mathcal{S}}$ to sweep over all record fields and abstract the field values. For vectors, $\widehat{\mathcal{S}}$ abstracts all the elements in the Scheme vector and joins them into an abstract value. The implementation is also able to abstract cyclic data in a meaningful way.

4.5 Precision and $\widehat{\textbf{Time}}$

The flow analysis as defined in Section 4.2 lacks a clear definition of the set of abstract times $\widehat{\textbf{Time}}$, but imposes only a loose condition a time abstraction: $\widehat{\textbf{Time}}$ must be finite to ensure termination.

The choice of a time abstraction is crucial for the precision and performance of the analysis. Recall that the flow analysis may fold many concrete states into a few abstract states. Consequently, the precision degrades.

$\widehat{\textbf{Time}}$, however, not only influences the size of the state space, but generally the number of abstract values, binding environments, and the merging of information in variable environments and stores. For example a variable environments maps tuples of variables and points in time to denotable values. Assume that $\widehat{\textbf{Time}}$ is a singleton set. Consequently, the variable environment does not distinguish variable values on basis of time: For each variable in the program the environment only maintains a single entry. That is, all values bound to a certain variable throughout a program run are merged in this single entry. For an abstraction distinguishing multiple abstract points in time, in contrast, the environment contains multiple entries for a single variable — each with a different point in time. Section 4.6 contains a concrete example for this behavior.

As depicted in the semantics, the flow analysis is decoupled from a particular implementation of $\widehat{\textbf{Time}}$. For the experiments conducted to study applications of the flow analysis in the transformational compiler I used the well-studied k-

[1]Note that $\widehat{\mathcal{S}}$ only abstracts procedures defined at top level, because the binding environment in these cases is known to be empty.

CFA time abstractions [*Shivers*, 1991]. Understanding the flow analysis results requires basic knowledge on these time abstractions. Thus, the following sections give a brief overview on the k-CFA analysis hierarchy.

4.5.1 k-CFA time abstractions

The time abstractions based on k-CFA use *call strings* to define points in time. In the execution of a program, the call sites stored on the call stack form the current call string. Thus, a call string denotes a sequence of call nodes, which are a sequence of syntax tree nodes. Semantically, a call string contains the *context* in which a call or evaluation takes place.

In his thesis, Shivers [*Shivers*, 1991] applies call strings to the flow analysis of programs to derive context information. This information forms the basis for a series of time abstractions with varying precision and performance — the k-CFA analysis hierarchy. As in the concrete execution of a program, the analysis maintains a string that lists the call sites visited during the abstract interpretation of a program. This context information is the foundation for $\widehat{\mathbf{Time}}$: The current time of two states in the abstract interpretation is equal if for both states the context information is equal. That is, the call strings are identical.

The k in k-CFA denotes how much of the available context information is used when comparing the context. In *1CFA* only the latest entry in the call string is relevant for the comparison of context information. For a *0CFA*, the analysis uses no context information at all. Thus, a 0CFA does not distinguish between different points in time: The set of abstract points in time is a singleton set.

0CFA is a *monovariant* analysis (or a *context-insensitive* analysis), and 1CFA is *polyvariant* or *context-sensitive*.

Before discussing the implications of these abstractions, consider the formal definitions for a 0CFA and 1CFA implementation of $\widehat{\mathbf{Time}}$. As mentioned, 0CFA is a singleton set, and thus the definition reads as follows:

$$\widehat{\mathbf{Time}}_0 = \{\widehat{t}_0\}$$

The 0CFA $\widehat{tick}$ function is a constant function that ignores the arguments $\widehat{t}$ and $\widehat{\varsigma}$ and just returns $\widehat{t}_0$:

$$\begin{aligned} &\widehat{tick}_0 && : (\widehat{\mathbf{Time}}_0 \times \mathbf{Eval}) \rightarrow \widehat{\mathbf{Time}}_0 \\ &\widehat{tick}_0(\widehat{t}, \widehat{\varsigma}) && = \widehat{t}_0 \end{aligned}$$

1CFA uses exactly one item of the call string as context information, that is a single call. Thus, the calls of a program form $\widehat{\mathbf{Time}}_1$:

$$\widehat{\mathbf{Time}}_1 = \mathtt{Call}$$

Recall that only transitions from *eval* states call $\widehat{tick}$ and pass the *eval* state as a supplementary argument to $\widehat{tick}$ (see 4.2). That is, the call that is subject of the eval call contains the context information. $\widehat{tick}_1$ simply extracts the call from the *eval* state:

$$\begin{aligned} &\widehat{tick}_1 && : (\widehat{\mathbf{Time}}_1 \times \mathbf{Eval}) \rightarrow \widehat{\mathbf{Time}}_1 \\ &\widehat{tick}_1(\widehat{t}, \widehat{\varsigma}) && = call \qquad \text{where } \widehat{\varsigma} = (call, \widehat{\beta}, \widehat{ve}, \widehat{t}) \end{aligned}$$

While it is possible to define time abstractions with $k > 1$ that involve more context information computing the analysis in these cases becomes impracticable. A 0CFA is computable in polynomial time. A 1CFA, however, already needs exponential time, and recent research implicates that k-CFA for greater k is also exponential in time [*van Horn and Mairson*, 2007]. Note, however, that even if 1CFA is exponential in time, computing 1CFA for realistic programs is often practical.

The 0CFA and 1CFA time abstractions are suitable for the applications considered in this dissertation. For further applications, however, it may be useful to employ different time abstractions. Note that the flow analysis as defined is not tied to k-CFA. It already lays the foundations for implementing more complex time abstractions such as the Cartesian product algorithm [*Agesen*, 1995] or polymorphic splitting [*Jagannathan and Wright*, 1998].

4.6 Flow analysis examples

This section exemplifies the flow analysis with two small examples, both computed using the implementation described in Chapter 6.

0CFA example Figure 4.6 shows the PreScheme factorial program in its intermediate representation and the state transitions of a 0CFA flow analysis for this program. Each node in the graph corresponds to a state in the visited set of the analysis. The states are numbered consecutively, the prefix letter "E" indicates an *eval* state and "A" an *apply* state. The table in Figure 4.6 contains detailed information about the states. The first column displays the name of the state. The "state" column shows the state with its components. Note that the table does not show the binding and variable environments explicitly, but assigns a name to each intermediate environment. Using the "changes" column and the graph the development of a particular environment are traceable. For *apply* states, the "changes" column shows how this application affects binding and variable environments. For *eval* states, this column shows which expression evaluates under which environments. The "transition" column denotes which transition rule applies for the state in this particular row.

As $\widehat{tick}$ is a constant function the "changes" column in the table does not list the calls to $\widehat{tick}$ that take place in each *eval* state.

Follow the graph from its root node A1. The state E2 evaluates the `letrec`, sets up the binding and variable environments and creates the *eval* state E3 which corresponds to the call in the body of `letrec`. In this state, the analysis evaluates the literal arguments to exact integers and the subsequent *apply* state A4 binds these values in the variable environment $\widehat{ve}_4$. Under this environment, the analysis evaluates the call to the `test` primop in E5. As specified by the $\widehat{\text{TestEval}}$ rule, this transition results into two *apply* states — one state for each branch of `test`.

The right branch starting at A6 represents the trivial case of the recursion. Lambda `c_33` calls a continuation to communicate the return value. The `unknown-return` primop for this call denotes that the compiler could not identify the continuation being called. The analysis, however, tracks the variable environment and thus discovers that the lambda returns via the `halt` continua-

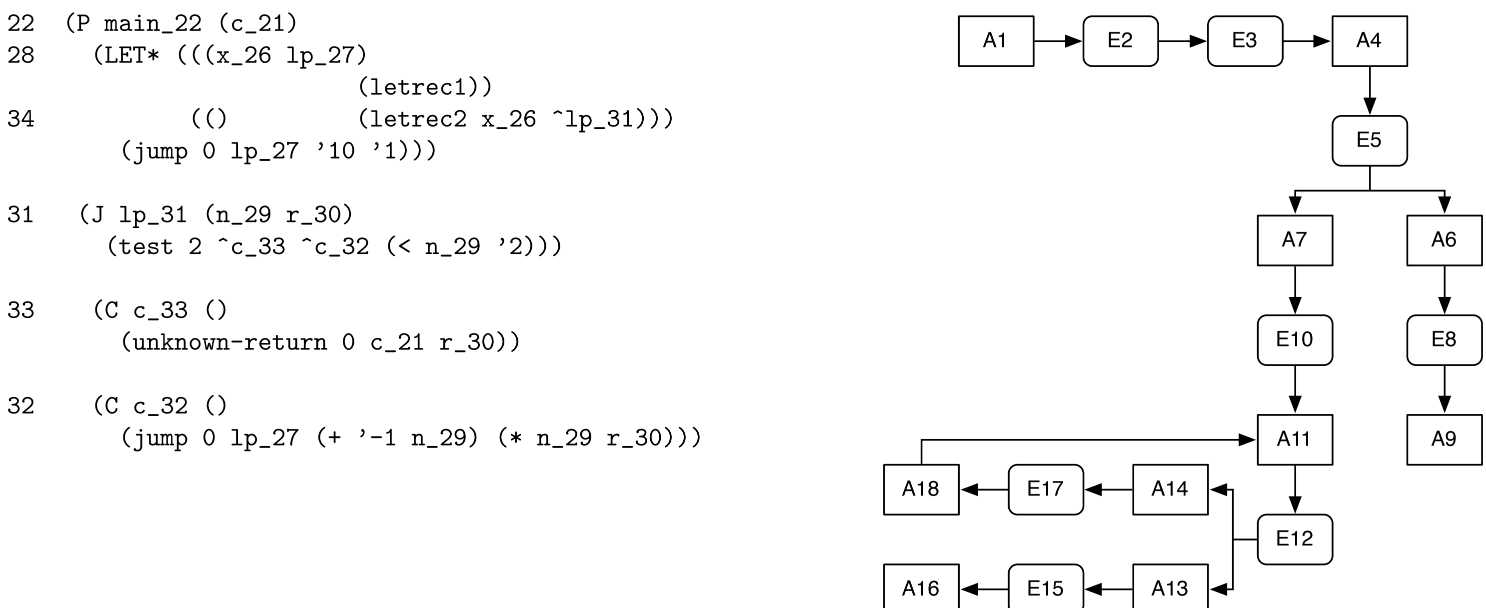

Figure 4.15: PreScheme `fact` program and the 0CFA state transition graph.

Name	State	Transition	Changes
A1	$(\{(\lambda^{\mathtt{P}}_{main}, \widehat{\beta}_1, \widehat{t_0})\}, \langle\rangle, \langle\{\mathtt{halt}\}\rangle, \widehat{ve}_1, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_2 = \widehat{\beta}_1[c_{21} \mapsto \widehat{t_0}], \widehat{ve}_2 = \widehat{ve}_1 \sqcup [(c_{21}, \widehat{t_0}) \mapsto \{\mathtt{halt}\}]$
E2	$(\texttt{(letrec1 \^{}c_28)}, \widehat{\beta}_2, \widehat{ve}_2, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{LetrecEval}}$	$\widehat{\beta}_3 = \widehat{\beta}_2[lp_{27} \mapsto \widehat{t_0}]$, $\widehat{ve}_3 = \widehat{ve}_2 \sqcup [(lp_{27}, \widehat{t_0}) \mapsto \{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}$
E3	$(\texttt{(jump lp_27 10 1)}, \widehat{\beta}_3, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{CCallEval}}$	$\widehat{proc} = \{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}$, $\widehat{d^*} = \langle\{10\}, \{1\}\rangle$
A4	$(\{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}, \langle\rangle, \langle\{10\}, \{1\}\rangle, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_5 = \widehat{\beta}_3[n_{29} \mapsto \widehat{t_0}, r_{30} \mapsto \widehat{t_0}]$, $\widehat{ve}_4 = \widehat{ve}_3 \sqcup [(n_{29}, \widehat{t_0}) \mapsto \{10\}, (r_{30}, \widehat{t_0}) \mapsto \{1\}]$
E5	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_29 2))}, \widehat{\beta}_5, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{TestEval}}$	$\widehat{b_0} = \{(\lambda^{\mathtt{C}}_{33}, \widehat{\beta}_5, \widehat{t_0})\}$, $\widehat{b_1} = \{(\lambda^{\mathtt{C}}_{32}, \widehat{\beta}_5, \widehat{t_0})\}$
A6	$(\{(\lambda^{\mathtt{C}}_{33}, \widehat{\beta}_5, \widehat{t_0})\}, \langle\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	
A7	$(\{(\lambda^{\mathtt{C}}_{32}, \widehat{\beta}_5, \widehat{t_0})\}, \langle\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	
E8	$(\texttt{(unknown-return c_21 r_30)}, \widehat{\beta}_5, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{CCallEval}}$	$\widehat{proc} = \{\mathtt{halt}\}$, $\widehat{d^*} = \langle\{1\}\rangle$
A9	$(\{\mathtt{halt}\}, \langle\rangle, \langle\widehat{\{1\}}\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	—	final state
E10	$(\texttt{(jump lp_27 (+ -1 n_29) (* n_29 r_30))}, \widehat{\beta}_5, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{CCallEval}}$	$\widehat{proc} = \{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}$, $\widehat{d^*} = \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle$
A11	$(\{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}, \langle\rangle, \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_6 = \widehat{\beta}_3[n_{29} \mapsto \widehat{t_0}, r_{30} \mapsto \widehat{t_0}]$, $\widehat{ve}_5 = \widehat{ve}_4 \sqcup [(n_{29}, \widehat{t_0}) \mapsto \{\top_{Int}\}, (r_{30}, \widehat{t_0}) \mapsto \{\top_{Int}\}]$
E12	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_29 2))}, \widehat{\beta}_6, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{TestEval}}$	$\widehat{b_0} = \{(\lambda^{\mathtt{C}}_{33}, \widehat{\beta}_6, \widehat{t_0})\}$, $\widehat{b_1} = \{(\lambda^{\mathtt{C}}_{32}, \widehat{\beta}_6, \widehat{t_0})\}$
A13	$(\{(\lambda^{\mathtt{C}}_{33}, \widehat{\beta}_6, \widehat{t_0})\}, \langle\rangle, \langle\rangle, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	
A14	$(\{(\lambda^{\mathtt{C}}_{32}, \widehat{\beta}_6, \widehat{t_0})\}, \langle\rangle, \langle\rangle, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	
E15	$(\texttt{(unknown-return c_21 r_30)}, \widehat{\beta}_6, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{CCallEval}}$	$\widehat{proc} = \{\mathtt{halt}\}$, $\widehat{d^*} = \langle\{1, \top_{Int}\}\rangle$
A16	$(\{\mathtt{halt}\}, \langle\rangle, \langle\{1, \top_{Int}\}\rangle, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	—	final state
E17	$(\texttt{(jump lp_27 (+ -1 n_29) (* n_29 r_30))}, \widehat{\beta}_6, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{CCallEval}}$	$\widehat{proc} = \{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}$, $\widehat{d^*} = \langle\{10, \top_{Int}\}, \{1, \top_{Int}\}\rangle$
A18	$(\{(\lambda^{\mathtt{C}}_{31}, \widehat{\beta}_3, \widehat{t_0})\}, \langle\rangle, \langle\{10, \top_{Int}\}, \{1, \top_{Int}\}\rangle, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t_0})$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_7 = \widehat{\beta}_3[n_{29} \mapsto \widehat{t_0}, r_{30} \mapsto \widehat{t_0}]$, $\widehat{ve}_6 = \widehat{ve}_5 \sqcup [(n_{29}, \widehat{t_0}) \mapsto \{10, \top_{Int}\}, (r_{30}, \widehat{t_0}) \mapsto \{1, \top_{Int}\}]$
E19	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_29 2))}, \widehat{\beta}_7, \widehat{ve}_6, \widehat{\sigma}_I, \widehat{t_0})$	—	already visited: $E12 \sqsubseteq E19$

Figure 4.16: PreScheme `fact` 0CFA state transitions

tion bound in state A1. Hence, A9 applies the `halt` continuation and therefore is a terminal state.

The left branch evaluates the recursive call to lambda `lp_31` in E10. Note that the evaluation of the trivial call arguments yields the inexact abstract integer value $\top_{Int}$. The variable environment $\widehat{ve}_5$ extended in the subsequent *apply* state A11 thus merges the abstract values for the variables `n_29` and `r_30` that accumulate the values of the recursion.

The abstract states A4 and A11 *apply* the same abstract closure at the same abstract point in time. A11 is, however, not an approximation of A4 because A11's argument vector $\widehat{d}^*$ is not weaker than A4's argument vector in the sense of $\sqsubseteq$. This also applies to the variable environments. Hence, the analysis schedules A11 for the unvisited queue and thereby evaluates the body of `lp_31` a second time.

Following the second evaluation of `lp_31` either leads to the final state A16 or another *eval* state for the evaluation of the recursive call. Consider A16 first. Like A9, this call applies the `halt` continuation, but this time also passes the value $\top_{Int}$ to the continuation. This is the expected behavior for a conservative analysis: A16 taken together with A9 gives the set of all values that may occur as the argument of the `halt` continuation.

The series of state transitions reaches the recursive call again in E17. E17, however is not an approximation of E10: While the constituents for the binding environment and time are weaker in the sense of $\sqsubseteq$, the variable environment is not. $\widehat{ve}_5 \not\sqsubseteq \widehat{ve}_4$: For $(n_29, \widehat{t_0}), (r_30, \widehat{t_0})$ the environment $\widehat{ve}_5$ includes the values $\top_{Int}$ but $\widehat{ve}_4$ does not. Hence, the analysis considers the recursive call a third time and applies the closure over `lp_31` again. The state E19 that evaluates the body of `lp_31`, however, is a state that is already member of the visited set. Comparing E19 with E12 yields that E19 is weaker than E12: The bindings environments are equal in the sense of $\sqsubseteq$ as well as the variable environments. Thus, the analysis stops here. In summary, the results of the analysis are:

- The continuation called at the call site `(unknown-return c_21 c_30)` is the `halt` continuation, because the continuation argument in the corresponding states E8 and E15 evaluates to {`halt`}. The argument is an integer.

- The arguments for the lambda expression `lp_31` are always integers.

1CFA example Figures 4.17 and 4.18 show the 1CFA flow analysis of the PreScheme factorial program. For this program, 1CFA computes a flow analysis with the same number of states as the 0CFA. Basically, the analysis proceeds as in the 0CFA case and computes the same result.

The 1CFA variant, however, distinguishes six different points in time. As discussed in Section 4.5, the 1CFA abstraction uses a call string of length one to measure time. That is, the last call visited during the state transition serves as the point in time. The following table shows the calls and the matching abbreviations $\widehat{t_i}$ used in the table of Figure 4.18:

$\widehat{t'}$	**Call site**
$\widehat{t_0}$	—
$\widehat{t_1}$	`(letrec1 ^c_28)`
$\widehat{t_2}$	`(jump lp_27 '10 '1)`
$\widehat{t_3}$	`(test ^c_33 ^c_32 (< n_29 '2))`
$\widehat{t_4}$	`(unknown-return c_21 r_30)`
$\widehat{t_5}$	`(jump lp_27 (+ '-1 n_29) (* n_29 r_30))`

Consider the *eval* states of the analysis. $\widehat{t'}$ denotes the value computed by $\widehat{tick}$ when applied to the current state and the current time. Note how the time is tied to the call node, not the *eval* state. The states E5 and E12 serve as examples. Both states correspond to the evaluation of exactly the same `test` primop call in the program. Thus, both states deal with the same piece of syntax and hence in both cases the time $\widehat{t'}$ is the same: $\widehat{t_3}$.

Unlike 0CFA, the 1CFA does not necessarily merge the arguments of unrelated calls. The program contains two calls to `lp_31`: The first call resides in the body of continuation lambda `c_34` with the arguments `10` and `1`. The second call is the body of continuation `c_32` and computes the arguments using two trivial calls.

Since the variable environment in a 0CFA contains at most one entry for a variable, multiple bindings coming from different calls merge into a single entry. Thus, the variable environment $\widehat{ve}_5$ of Figure 4.6 has the following entries:

$$
\begin{array}{ll}
(c_{21}, \widehat{t_0}) & \mapsto \{\mathtt{halt}\} \\
(lp_{27}, \widehat{t_0}) & \mapsto \{(\lambda^{\mathtt{c}}_{31}, \widehat{\beta}_4, \widehat{t_0})\} \\
(n_{29}, \widehat{t_0}) & \mapsto \{10, \top_{Int}\} \\
(r_{30}, \widehat{t_0}) & \mapsto \{1, \top_{Int}\}
\end{array}
$$

Note that here the analysis merges the values for `n_29` and `r_30` even though the values originate from different calls. The 1CFA variable environment $\widehat{ve}_4$, from Figure 4.18 for the same state (A11), however, distinguishes both values on basis of their different times $\widehat{t_2}$ and $\widehat{t_5}$:

$$
\begin{array}{ll}
(c_{21}, \widehat{t_0}) & \mapsto \{\mathtt{halt}\} \\
(lp_{27}, \widehat{t_1}) & \mapsto \{(\lambda^{\mathtt{c}}_{31}, \widehat{\beta}_4, \widehat{t_1})\} \\
(n_{29}, \widehat{t_2}) & \mapsto \{10\} \\
(r_{30}, \widehat{t_2}) & \mapsto \{1\} \\
(n_{29}, \widehat{t_5}) & \mapsto \{\top_{Int}\} \\
(r_{30}, \widehat{t_5}) & \mapsto \{\top_{Int}\}
\end{array}
$$

Now, consider the states evaluating the calls to `lp_31`: the states E3, E10, and E17. In E3 the time advances to $\widehat{t_2}$, in E10 and E17 to $\widehat{t_5}$. Thus, the successor states update the binding environments for these times. In A4, the binding environment $\widehat{\beta}_5$ reads as follows:

$$
\begin{array}{ll}
c_{21} & \mapsto \widehat{t_0} \\
lp_{27} & \mapsto \widehat{t_1} \\
n_{29} & \mapsto \widehat{t_2} \\
r_{30} & \mapsto \widehat{t_2}
\end{array}
$$

Name	State	Transition	Changes
A1	$(\{(\lambda^{\mathsf{P}}_{main}, \widehat{\beta}_2, \widehat{t}_0)\}, \langle\rangle, \langle\{\texttt{halt}\}\rangle, \widehat{ve}_I, \widehat{\sigma}_I, \widehat{t}_0)$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_3 = \widehat{\beta}_2[c_{21} \mapsto \widehat{t}_0]$, $\widehat{ve}_1 = \widehat{ve}_I \sqcup [(c_{21}, \widehat{t}_0) \mapsto \{\texttt{halt}\}]$
E2	$(\texttt{(letrec1 \^{}c_28)}, \widehat{\beta}_3, \widehat{ve}_1, \widehat{\sigma}_I, \widehat{t}_0)$	$\widehat{\text{LetrecEval}}$	$\widehat{t'} = \widehat{t}_1$, $\widehat{\beta}_4 = \widehat{\beta}_3[lp_{27} \mapsto \widehat{t}_1]$, $\widehat{ve}_2 = \widehat{ve}_1 \sqcup [(lp_{27}, \widehat{t}_1) \mapsto \{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}]$
E3	$(\texttt{(jump lp_27 10 1)}, \widehat{\beta}_4, \widehat{ve}_2, \widehat{\sigma}_I, \widehat{t}_1)$	$\widehat{\text{CCallEval}}$	$\widehat{t'} = \widehat{t}_2$, $\widehat{proc} = \{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}$, $\widehat{d^*} = \langle\{1\}, \{10\}\rangle$
A4	$((\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1), \langle\{10\}, \{1\}\rangle, \langle\rangle, \widehat{ve}_2, \widehat{\sigma}_I, \widehat{t}_2)$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_5 = \widehat{\beta}_4[n_{29} \mapsto \widehat{t}_2, r_{30} \mapsto \widehat{t}_2]$, $\widehat{ve}_3 = \widehat{ve}_2 \sqcup [(n_{29}, \widehat{t}_2) \mapsto \{10\}, (r_{30}, \widehat{t}_2) \mapsto \{1\}]$
E5	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_28 2))}, \widehat{\beta}_5, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_2)$	$\widehat{\text{TestEval}}$	$\widehat{t'} = \widehat{t}_3$, $\widehat{b}_0 = \{(\lambda^{\mathsf{C}}_{33}, \widehat{\beta}_5, \widehat{t}_3)\}$, $\widehat{b}_1 = \{(\lambda^{\mathsf{C}}_{32}, \widehat{\beta}_5, \widehat{t}_3)\}$
A6	$(\{(\lambda^{\mathsf{C}}_{33}, \widehat{\beta}_5, \widehat{t}_3)\}, \langle\rangle, \langle\rangle, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{ApplyClos}}$	
A7	$(\{(\lambda^{\mathsf{C}}_{32}, \widehat{\beta}_5, \widehat{t}_3)\}, \langle\rangle, \langle\rangle, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{ApplyClos}}$	
E8	$(\texttt{(unknown-return c_21 r_30)}, \widehat{\beta}_5, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{CCallEval}}$	$\widehat{t'} = \widehat{t}_4$, $\widehat{proc} = \{\texttt{halt}\}$, $\widehat{d^*} = \langle\{1\}\rangle$
A9	$(\{\texttt{halt}\}, \langle\{1\}\rangle, \langle\rangle, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_4)$	—	final state
E10	$(\texttt{(jump lp_27 (+ -1 n_29) (* n_29 r_30))}, \widehat{\beta}_5, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{CCallEval}}$	$\widehat{t'} = \widehat{t}_5$, $\widehat{proc} = \{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}$, $\widehat{d^*} = \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle$
A11	$(\{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}, \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle, \langle\rangle, \widehat{ve}_3, \widehat{\sigma}_I, \widehat{t}_5)$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_6 = \widehat{\beta}_4[n_{29} \mapsto \widehat{t}_5, r_{30} \mapsto \widehat{t}_5]$, $\widehat{ve}_4 = \widehat{ve}_3 \sqcup$ $[(n_{29}, \widehat{t}_5) \mapsto \{\top_{Int}\}, (r_{30}, \widehat{t}_5) \mapsto \{\top_{Int}\}]$
E12	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_29 2))}, \widehat{\beta}_6, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_5)$	$\widehat{\text{TestEval}}$	$\widehat{t'} = \widehat{t}_3$, $\widehat{b}_0 = \{(\lambda^{\mathsf{C}}_{33}, \widehat{\beta}_6, \widehat{t}_3)\}$, $\widehat{b}_1 = \{(\lambda^{\mathsf{C}}_{32}, \widehat{\beta}_6, \widehat{t}_3)\}$,
A13	$(\{(\lambda^{\mathsf{C}}_{33}, \widehat{\beta}_6, \widehat{t}_3)\}, \langle\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{ApplyClos}}$	
A14	$(\{(\lambda^{\mathsf{C}}_{32}, \widehat{\beta}_6, \widehat{t}_3)\}, \langle\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{ApplyClos}}$	
E15	$(\texttt{(unknown-return c_21 r_30)}, \widehat{\beta}_6, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{CCallEval}}$	$\widehat{t'} = \widehat{t}_4$, $\widehat{proc} = \{\texttt{halt}\}$, $\widehat{d^*} = \langle\{\top_{Int}\}\rangle$
A16	$(\{\texttt{halt}\}, \langle\{\top_{Int}\}\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_4)$	—	final state
E17	$(\texttt{(jump lp_27 (+ -1 n_29) (* n_29 r_30))}, \widehat{\beta}_6, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_3)$	$\widehat{\text{CCallEval}}$	$\widehat{t'} = \widehat{t}_5$, $\widehat{proc} = \{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}$, $\widehat{d^*} = \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle$
A18	$(\{(\lambda^{\mathsf{C}}_{31}, \widehat{\beta}_4, \widehat{t}_1)\}, \langle\{\top_{Int}\}, \{\top_{Int}\}\rangle, \langle\rangle, \widehat{ve}_4, \widehat{\sigma}_I, \widehat{t}_5)$	$\widehat{\text{ApplyClos}}$	$\widehat{\beta}_7 = \widehat{\beta}_4[n_{29} \mapsto \widehat{t}_5, r_{30} \mapsto \widehat{t}_5]$, $\widehat{ve}_5 = \widehat{ve}_4 \sqcup [(n_{29}, \widehat{t}_5) \mapsto \{\top_{Int}\}, (r_{30}, \widehat{t}_5) \mapsto \{\top_{Int}\}]$
E19	$(\texttt{(test \^{}c_33 \^{}c_32 (< n_29 2))}, \widehat{\beta}_7, \widehat{ve}_5, \widehat{\sigma}_I, \widehat{t}_5)$	—	already visited: $E12 \sqsubseteq E19$

Figure 4.17: PreScheme `fact` program and the 1CFA state transition graph.

```
22 (P main_22 (c_21)
28   (LET* (((x_26 lp_27)
               (letrec1))
34          (() (letrec2 x_26 ^lp_31)))
      (jump 0 lp_27 '10 '1)))

31  (J lp_31 (n_29 r_30)
      (test 2 ^c_33 ^c_32 (< n_29 '2)))

33   (C c_33 ()
       (unknown-return 0 c_21 r_30))

32   (C c_32 ()
       (jump 0 lp_27 (+ '-1 n_29) (* n_29 r_30)))
```

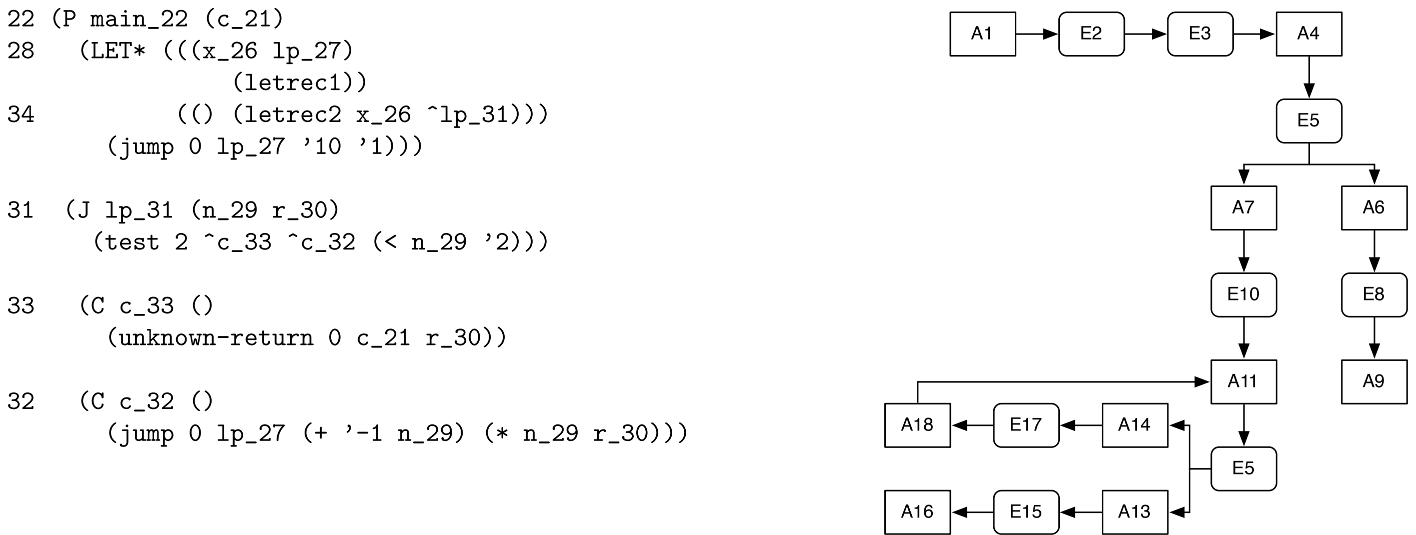

Figure 4.18: PreScheme `fact` 1CFA state transitions

The states A11 and A18, however, create binding environments $\widehat{\beta}_6$ and $\widehat{\beta}_7$ which map the binding time for `n_29` and `r_30` to $\widehat{t}_5$. That is, the subsequent evaluation of variables taking place in the successor *eval* states access the proper entries of the variable environment and do not confuse the values with values of other dynamic instances of the call.

Chapter 5

Executable Semantic Model

This chapter develops an *executable model* of the semantics which, I used to test, inspect, trace, and visualize the semantics described in the previous chapters.

The actual implementation (see Chapter 6) has some properties that make it hard to trace the analysis and judge whether the semantics and the behavior of the analysis are synchronous. The executable model of the semantics, however, is easy to trace and visualizes the mode of operation. Each transition rule and metafunction defined in the semantics has a counterpart in the model which makes it easy to validate that the model exactly works like the mathematical formulation. The mode of operation is as follows: A test case for the model considers a program pr and computes $\mathcal{V}(pr)$ and $\widehat{\mathcal{V}}(pr)$. Then, the model checks whether Theorem 4.3 holds. That is, it checks whether each state of a concrete program execution has a counterpart in $\widehat{\mathcal{V}}$ such that $\mathcal{R}$ for these states yields true.

The implementation differs from the model in the following aspects:

- To improve the performance, the implementation uses complex data structures to represent the semantic domains.
- The implementation is tightly coupled to the compiler and setting up programs as test cases is complicated.
- The results of an analysis are stored in data structures that are inconvenient to inspect and validate.

The PLT Redex tool [*Matthews et al.*, 2004] facilitates executing semantics formulated as a term rewrite system. I use PLT Redex to model the semantics. The model consists of the following parts:

- A grammar for the intermediate language.
- Implementations of all concrete domains as terms and a formulation of the concrete state transition relation $\rightarrow$ as a reduction relation on terms.
- Implementations of all abstract domains as terms and a formulation of the abstract state transition relation $\widehat{\rightarrow}$ as a reduction relation on terms.
- A straightforward implementation of the correctness relation $\mathcal{R}$.

- An extensive test suite for all reduction relations and metafunctions.
- Code for exporting the state transitions in $\mathcal{V}$ and $\widehat{\mathcal{V}}$ in form of a GML file [*Himsolt*, 1995] — a file format for describing graphs — that allows visualizing state transitions with an external program.

5.1 Example trace

Before turning to some aspects of the implementation of the executable semantics, consider the following example: Figure 5.1 shows the evaluation of 5! in PreScheme (see Figure 4.6 for the source code) in both the concrete and the abstract semantics. Both graphs in Figure 5.1 were produced by the executable model: Rectangle nodes depict the states visited by the concrete evaluation and oval nodes depict abstract states. Grey nodes denote *apply* states (again, *apply* state names start with the letter A) and white nodes are *eval* states (their names start with the letter E). Dashed lines illustrate the correctness relation: For every abstract state $\widehat{\varsigma}$ connected by a dashed line to a concrete state ς the correctness relation $\varsigma \; \mathcal{R} \; \widehat{\varsigma}$ holds.

The abstract states in Figure 5.1 correspond to the states shown in the 1CFA example of the previous section (see the table in Figure 4.18). Both runs start with the initial state A1 that applies the `main` procedure to an empty argument vector and `halt` — the flow analysis and the program run start in sync.

Recall that lambda `lp_31` is the core of the loop that computes the factorial. The body of this lambda checks whether the end of the recursion has been reached and either calls a continuation that in turn calls `lp_31` recursively or a continuation that returns the factorial number.

The state table in Figure 4.18 shows that the flow analysis contains three *apply* states for `lp_31`: A4, A11, and A18. In the real program run, however, there must be five applications of `lp_31` for the factorial of five. The states are A5, A9, A13, A17, and A21.

Consider the control flow starting at the abstract state A4 — this is the first application of `lp_31`. Here, the flow analysis knows the exact values for the literal integer arguments. Then, the analysis evaluates the body of `lp_31` under the updated environments. This happens in states E5 and E6, respectively. Note that the abstract evaluation of the `test` primop follows both branches and hence E5 has two subsequent states. Its concrete counterpart E6 only calls the continuation for the alternative branch. That is, the abstract states A6, E8 and A9 characterize a situation that does not take place in a concrete program run.

The second application of `lp_31` in the flow analysis starts with A11. The arguments for this application derive from the evaluation of primops and hence contain $\top_{Int}$. Therefore, A11 is more abstract than the first *apply* state for `lp_31`. In fact, A11 is a correct abstraction for all applications of `lp_31` that occur in the concrete program run. This also applies to all abstract states derived from A11: The states that characterize the evaluation of the alternative branch E12, A14, E17, and A18 are the abstract counterparts for the recursive calls to `lp_31`.

In the concrete *eval* state E22 `test` evaluates to true. This ends the recursion and returns the computed value by applying the `halt` continuation (abstract *apply* state A16 and concrete state A25).

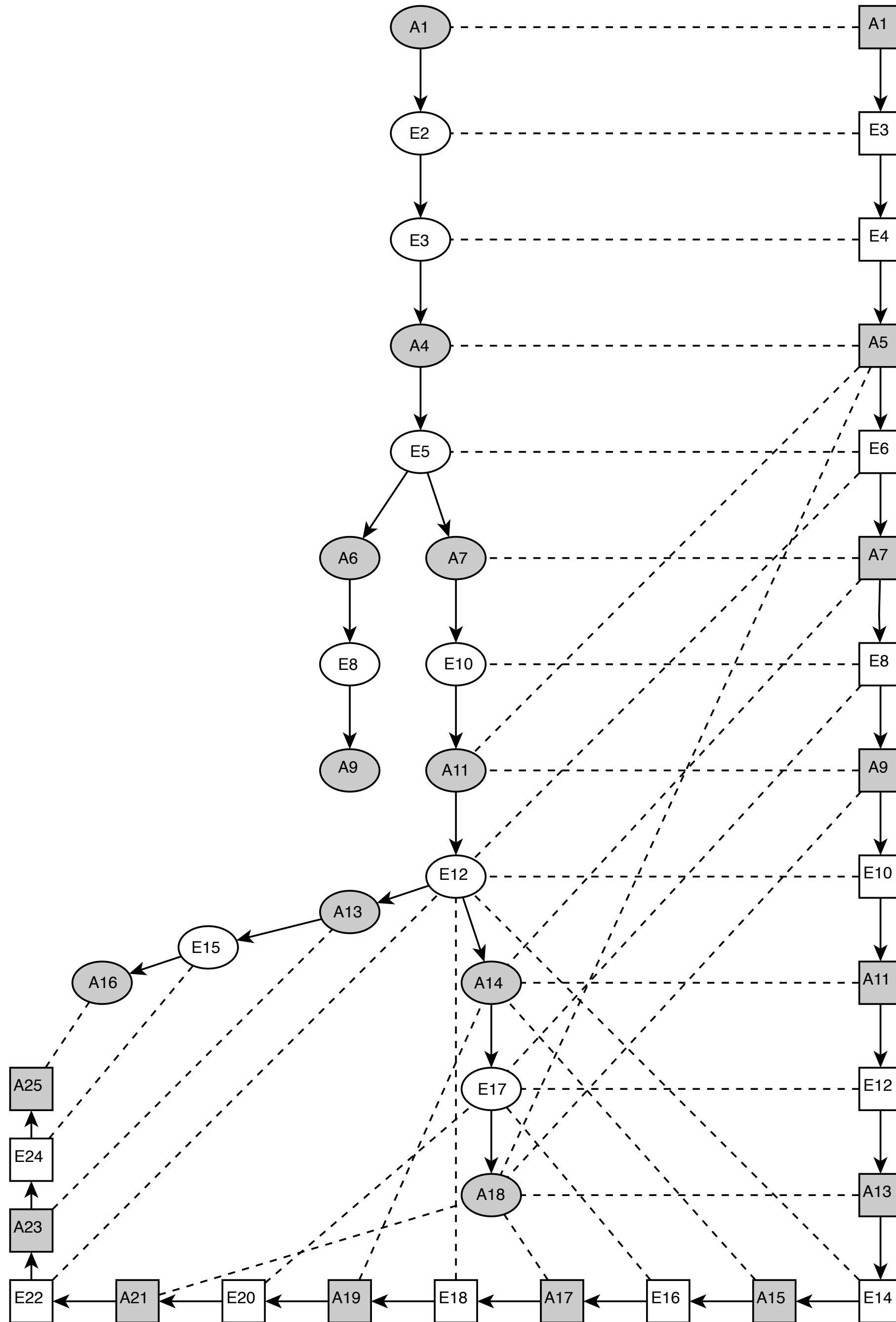

Figure 5.1: Semantic correctness for the PreScheme factorial program

Finally, note that this visualization makes checking the semantic correctness easy: Each concrete state has a dashed connection to an abstract state.

5.2 Implementation with PLT Redex

The implementation of the executable semantic model consists of about 2,000 lines of Scheme code, more than half of which are test cases. The grammar of the intermediate language as specified in Figure 2.2 translates directly into a grammar description for PLT Redex. This section shows the complete code of the grammar definition interweaved with explanations.

5.2.1 Grammar definition

The special form `language` defines a grammar as a list of productions. Each production is a pair consisting of a non-terminal name and a S-expressions for the right-hand side of the production. Additional S-expressions are treated as alternatives. Note that terms in PLT Redex always have the form of S-expressions. The definition of the grammar starts as follows:

```
(define cps-language
  (language
   ;;; labels
   (label number)
   ;;; each state has a unique id
   (id    number)
   ;;; variables
   (var-name  (variable-except call trivcall
                         Lambda-c Lambda-u Lambda-j
                         VEnv BEnv Time
                         Clos halt
                         Apply Eval
                         Literal))
   (var  (LVar var-name))
   (gvar (GVar var-name))
```

The label of a term is an unique integer that identifies the term unambiguously. The labels establish the basis for an intensional equality predicate.[1]

I will discuss the production `id` in context of the syntax for states. The definition above uses the two special non-terminal symbols `number` and `variable-except` provided by Redex. `Number` matches all numbers and `variable-except` matches all symbols except the ones specified. Here, `variable-except` prevents that the program contains one of the specified reserved names as a variable name.

The grammar of Figure 2.2 has disjoint sets for local and global variable names. The model, however, uses one set for variable names and distinguishes references by embedding the variable name into the forms `(LVar ...)` and `(GVar ...)`.

The next part of the grammar specifies the terms of the CPS intermediate language:

[1]The implementation (see Chapter 6) compares the nodes in the abstract syntax tree using pointer comparison to implement intensional equality.

```
(literal   (Literal number) (Literal true) (Literal false)
           (Literal string))

(prim-proc-call call tail-call unknown-call unknown-tail-call)
(prim-cont-call return unknown-return jump let)
(prim-lang-call + - * < >)
(prim-global    global-ref global-set!)

(lam-cont   (Lambda-c label (var-name ...) pcall))
(lam-user   (Lambda-u label (var-name ...) pcall))
(lam-jump   (Lambda-j label (var-name ...) pcall))

(exp      lam-cont lam-jump lam-user var gvar literal call-triv)
(exp-cont lam-cont lam-jump var gvar literal call-triv)
(exp-user lam-user var gvar literal call-triv)

(pcall     call-user call-cont)
(call-user (Call label prim-proc-call exp ...)
           (Call label prim-lang-call exp ...)
           (Call label prim-global exp ...)
           (Call label letrec1
                 (Lambda-c label (var-name ...)
                    (Call label letrec2 lam-cont exp ...))))
(call-cont (Call label prim-cont-call exp ...)
           (Call label test lam-cont lam-cont exp))
(call-triv (Trivcall label prim-lang-call exp ...))
```

The above defines the syntax for literals, the names of the primops, lambda expressions, call expressions, and expressions. Only boolean, number, and string literals are supported in the model. Also, the set of primops is limited to the core primops for calls and the basic arithmetic and comparison primops. The ellipsis operator `...` specifies that the last symbol before `...` may occur zero or more times.

Lambda expressions and calls are lists that have a tag and a label as their first elements. The tag for lambda expressions is a terminal symbol that denotes whether the expression is a continuation, jump, or user-defined lambda. For calls there is only the single tag `Call` as the primop identifies the category the call belongs to.

Figure 5.2 shows the definition for the procedure `lp_31` used in the example of the previous section (see Figure 4.17 for the complete source code of the PreScheme factorial program). For this example, I created the code manually. However, connecting the front end of the transformational compiler to the semantic model is straightforward. The special form `term` (provided by Redex) introduces a term and behaves similar to the Scheme `quasiquote` functionality: Comma escapes the quote and evaluates the expression followed by comma.

The presentation continues with the part of the grammar that defines the syntax for terms representing the domains of the concrete semantics. The first part deals with the binding environment (`benv`), the variable environment (`venv`), and the store (`store`):

```
(benv-entries (var-name time) ...)
(benv         (BEnv benv-entries ...))
```

```
(define ps-factorial-c32
  (term (Lambda-c 32 ()
          (Call 321 jump (LVar lp27)
                         (Trivcall 322 + (Literal -1) (LVar n29))
                         (Trivcall 323 * (LVar n29) (LVar r30))))))

(define ps-factorial-c33
  (term (Lambda-c 33 ()
          (Call 331 unknown-return (LVar c21) (LVar r30)))))

(define ps-factorial-lp31
  (term (Lambda-j 31 (n29 r30)
          (Call 310 test
                    ,ps-factorial-c33
                    ,ps-factorial-c32
                    (Trivcall 311 < (LVar n29) (Literal 2))))))

(define ps-factorial-main22
  (term (Lambda-c 22 (c21)
          (Call 220 letrec1
             (Lambda-c 221 (lp27)
                 (Call 222 letrec2
                      (Lambda-c 223 ()
                         (Call 224 jump (LVar lp27) (Literal 5) (Literal 1)))
                      ,ps-factorial-lp31))))))
```

Figure 5.2: Representing a program in the executable semantics

```
(venv-entries (var-name time value) ...)
(venv         (VEnv venv-entries ...))

(store-entries (ref value))
(store         (Store store-entries ...))
```

The corresponding domains **BEnv**, **VEnv**, and **Store** in the semantics are function domains. The model represents the functions as terms containing sequences of tuples. Each tuple consists of a key and a value. Note that it would also be possible to represent the functions as Scheme procedures: This, however, has the disadvantage that a trace of the reduction relation would contain opaque Scheme values and the result would be less traceable. The next section shows the metafunctions — functions from terms to terms — that operate on the terms representing functions.

A number wrapped in the form `(Time ...)` represents the elements of the domain **Time**:

```
(time  (Time number))
```

The following code defines the rules for terms representing basic values, procedure values, references, and locations:

```
(bool         (Bool true) (Bool false))
(int          (Int number))
```

```
(closure      (Clos lam-cont benv time)
              (Clos lam-user benv time)
              (Clos lam-jump benv time))
(proc         closure halt)

(value        proc int bool ref)

(ref          (Ref loc literal))
(loc          hloc gloc)
(hloc         (HLoc label time))
(gloc         (GLoc var-name))
```

The model supports boolean values and integers as basic value types. That is, the domain $\mathbf{Bas}_{Lang}$ has two summands:

$$\mathbf{Bas}_{Lang} = \mathbf{Boolean} + \mathbf{Integer}$$

All value terms are lists with a tag that identifies the domain the value belongs to.

Finally, the terms representing the states from the domain **State** can be defined:

```
(eval         (Eval id id call-user benv venv store time)
              (Eval id id call-cont benv venv store time))
(apply        (Apply id id proc (value ...) (value ...) venv store
                     time))
(state        eval apply)
```

A state is a list with a tag for the state type and all components of the state tuples. *Eval* and *apply* states have two additional components that contain numeric ids. The first number identifies the state and the second number is the id of its predecessor. The ids simplify tracing transitions and facilitate exporting the transitions as a directed graph.

This completes the grammar rules necessary to model the concrete semantics. The next part of the grammar definition considers the abstract semantics:

```
(abenv-entries (var-name atime) ...)
(abenv         (ABEnv abenv-entries ...))

(avenv-entries (var-name atime avalue) ...)
(avenv         (AVEnv avenv-entries ...))

(astore-entries (ARef avalue) ...)
(astore         (AStore astore-entries ...))

(atime         (ATime number))

(abool         (ABool bot) (ABool true) (ABool false) (ABool top))
(aint          (AInt bot) (AInt number) (AInt top))
(aclosure      (AClos lam-cont abenv atime)
               (AClos lam-user abenv atime)
               (AClos lam-jump abenv atime))
(aproc         aclosure halt)
(dvalue        aproc abool aint aref)
```

```
   (avalue       (AValue dvalue ...))

   (aref         (ARef aloc literal))
   (aloc         ahloc agloc)
   (ahloc        (AHLoc label time))
   (agloc        (AGLoc gvar))

   (aeval        (AEval id id call-user abenv avenv astore atime)
                 (AEval id id call-cont abenv avenv astore atime))
   (aapply       (AApply id id avalue (avalue ...) (avalue ...) avenv
                         astore atime))
   (astate       aeval aapply)))
```

The domains of the abstract semantics are modeled like their concrete counterparts. Note that productions for basic value domains $\widehat{\textbf{Integer}}$ and $\widehat{\textbf{Boolean}}$ now contains terms that represent the least and greatest values of the lattice.

The production `dvalue` models a denotable values. The terms matching `avalue` represent the abstract values of the domain $\widehat{\mathbf{D}}$. That is, a set of denotable values represented as a sequence.

This completes the definition of the grammar. The following section shows the metafunctions that operate on terms just described.

5.2.2 Metafunctions

Metafunctions are functions from terms to terms. The model uses metafunctions heavily to accomplish the following operations:

- extract parts of a CPS term.
- search and update the binding environment, the variable environment, and the store.
- join abstract values with $\sqcup$.
- implement the argument evaluation functions $\mathcal{A}$ and $\widehat{\mathcal{A}}$, the literal evaluation functions $\mathcal{K}_{Lang}$ and $\widehat{\mathcal{K}}_{Lang}$, and the primop evaluation functions $\mathcal{P}_{Lang}$ and $\widehat{\mathcal{P}}_{Lang}$.
- implement the approximation relation $\sqsubseteq$.
- implement the correctness relation $\mathcal{R}$.

This section shows most of the metafunctions used in the executable model.

The following metafunctions operate on CPS terms and extract parts of terms. `Call-cont-exp` extracts the expression that evaluates to the continuation from a call node:

```
(define-metafunction call-cont-exp
  cps-language
  ((Call label_1 prim-proc-call_1 exp_1 exp_2 ...)
   exp_1)
  ((Call label_1 prim-lang-call_1 exp_1 exp_2 ...)
   exp_1)
  ((Call label_1 test lam-cont_1 lam-cont_2 exp_1)
   (lam-cont_1 lam-cont_2))
```

```
  ((Call label_1 prim-cont-call_1 exp_1 exp_2 ...)
   exp_1))
```

The syntax for defining metafunctions resembles the Scheme macro system. A metafunction is a list of rules, where each rule consists of a left-hand side — the pattern — and a right-hand side that is the expression to evaluate if the pattern matches. The code for `call-cont-exp` consists of four rules. The Redex pattern language supports ellipses and binding matching subpatterns to variables as indicated by the suffixes _1, _2, and so on. In the first rule the pattern is `(Call label_1 prim-proc-call_1 exp_1 exp_2 ...)` and the right-hand side is the term `exp_1`. That is, if this pattern matches the metafunction returns the term matching the subpattern `exp_1`. Usually the right-hand side is a term, however, the programmer may use unquote `,` and specify a Scheme expression that evaluates to a term instead. Furthermore, there is a similar metafunction `call-op-exp` that extracts the expression in operator position.

The next metafunction, `call-args-exps`, extracts the argument list of a call node omitting the expressions in continuation and operator position:

```
(define-metafunction call-arg-exps
  cps-language
  ((Call label_1 prim-proc-call_1 exp_1 exp_2 exp_3 ...)
   (exp_3 ...))
  ((Call label_1 prim-lang-call_1 exp_1 exp_2 ...)
   (exp_2 ...))
  ((Call label_1 prim-cont-call_1 exp_1 exp_2 ...)
   (exp_2 ...)))
```

There are three kinds of lambda expressions and the following metafunctions provide a common way of extracting the body and parameter list:

```
(define-metafunction lambda-body
  cps-language
  ((Lambda-c label_1 (var-name_1 ...) pcall_1)
   pcall_1)
  ((Lambda-u label_1 (var-name_1 ...) pcall_1)
   pcall_1)
  ((Lambda-j label_1 (var-name_1 ...) pcall_1)
   pcall_1))

(define-metafunction lambda-params
  cps-language
  ((Lambda-c label_1 (var-name_1 ...) pcall_1)
   (var-name_1 ...))
  ((Lambda-u label_1 (var-name_1 ...) pcall_1)
   (var-name_1 ...))
  ((Lambda-j label_1 (var-name_1 ...) pcall_1)
   (var-name_1 ...)))
```

To compare CPS nodes by their label the model uses the comparison predicates `call=?` and `lambda=?`. Here is the definition of `call=?`:

```
(define-multi-args-metafunction call=?
  cps-language
  (((Call label_1 any ...) (Call label_1 any ...))
   #t)
```

```
  ((any any)
   #f))
```

This definition uses `define-multi-args-metafunction` a variant of `define-metafunction` that allows the definition of metafunctions with multiple arguments. `Lambda=?` is analog to `call=?`.

Variable names and literals do not have a label and the corresponding metafunctions `var=?` and `literal=?`, and thus these functions compare the whole term:

```
(define-multi-args-metafunction var=?
  cps-language
  ((var-name_1 var-name_1) #t)
  ((var-name_1 var-name_2) #f))
```

The binding environment is a sequence of tuples. Each tuple is called an entry and consists of a variable and a point in time. The following metafunctions search a binding environment for a given variable:

```
(define-multi-args-metafunction benv-key=?
  cps-language
  ((var-name_1 (var-name_2 time_1))
   ,(and (term (var=? var-name_1 var-name_2))
         (term time_1))))

(define-multi-args-metafunction benv-lookup
  cps-language
  (((BEnv benv-entries_1 ...) (LVar var-name_1))
   ,(my-find (lambda (entry)
               (term (benv-key=? var-name_1 ,entry)))
             (term (benv-entries_1 ...)))))
```

These metafunctions use a comma to unquote and evaluate a Scheme expression that generates the term to return. `Benv-key=?` checks whether a given entry matches the variable `var-name_1`. `Benv-lookup` searches the binding environment for an entry using the auxiliary Scheme function `my-find`, which is defined as follows:

```
(define (my-find p lst)
  (let lp ((lst lst))
    (if (null? lst)
        #f
        (or (p (car lst))
            (lp (cdr lst))))))
```

The semantics uses $\beta(v)$ to retrieve the binding time of v from the binding environment β. In the model this is expressed as `(benv-lookup benv (LVar v))` where `benv` is a representation for β.

Here is the metafunction that implements the update of a binding environment:

```
(define-multi-args-metafunction benv-update
  cps-language
  (((BEnv benv-entries_1 ...) (LVar var-name_1) time_1)
   ,(if (term (benv-lookup (BEnv benv-entries_1 ...) (LVar var-name_1)))
        (term (BEnv ,@(map (lambda (entry)
```

```
                            (if (term (benv-key=? var-name_1 ,entry))
                                (term (var-name_1 time_1))
                                entry))
                          (term (benv-entries_1 ...)))))
        (term (BEnv ,@(append (term (benv-entries_1 ...))
                             (term ((var-name_1 time_1)))))))))
```

The term $\beta[\mathtt{x} \mapsto t_0]$ in mathematical notation translates to `(benv-update benv (LVar x) (Time 0))` in the model. To update the binding environment `benv-update` creates a new term that represents the updated environment. The update proceeds as follows: If the binding environment already contains an entry for the variable `x` (the variable to update) `benv-lookup` uses `map` to traverse all entries and replace the entry for `x` — leaving the other entries untouched. If the binding environment has no entry `x`, `benv-update` appends a new entry.

The metafunctions for variable environments and store work analogously, therefore I only present the metafunctions for variable environments:

```
(define-multi-args-metafunction venv-key=?
  cps-language
  ((var-name_1 time_1 (var-name_2 time_2 value_1))
   ,(and (term (var=? var-name_1 var-name_2))
         (term (time=? time_1 time_2))
         (term value_1))))

(define-multi-args-metafunction venv-lookup
  cps-language
  (((VEnv venv-entries_1 ...) (LVar var-name_1) time_1)
   ,(my-find (lambda (entry)
               (term (venv-key=? var-name_1 time_1 ,entry)))
             (term (venv-entries_1 ...)))))
```

Like binding environments, variable environments are sequences of entries. Each entry is a tuple that consists of a variable name and a point in time that together form key and a value. The metafunction `venv-key=?` compares the key of an entry to a given variable name and time. `Venv-key?` returns the value stored in this entry if the key matches or returns `#f` if the key does not match. `Venv-lookup` uses `my-find` to search all entries.

In the semantics, $ve(v, \beta(v))$ returns the value of the variable v in the variable environment ve and the binding environment β. This translates to

```
(venv-lookup venv (LVar v) (benv-lookup benv (LVar v)))
```

in the executable model.

`Venv-update` updates a variable environment and relates to $ve[(v, t) \mapsto d]$ in the semantics:

```
(define-multi-args-metafunction venv-update
  cps-language
  (((VEnv venv-entries_1 ...) (LVar var-name_1) time_1 value_1)
   ,(if (term (venv-lookup (VEnv venv-entries_1 ...) (LVar var-name_1) time_1))
        (term (VEnv ,@(map (lambda (entry)
                            (if (term (venv-key=? var-name_1 time_1 ,entry))
                                (term (var-name_1 time_1 value_1))
                                entry))
                          (term (venv-entries_1 ...)))))
```

```
        (term (VEnv ,@(append (term (venv-entries_1 ...))
                              (term ((var-name_1 time_1 value_1))))))))
```

The mode of operation is similar to benv-update. That is, venv-update generates a new term that represents the updated environment.

The transition rules usually update the variable environment for all call arguments. Therefore venv-update/params folds venv-update over the arguments:

```
(define-multi-args-metafunction venv-update/params
  cps-language
  ((venv_1 (var-name_1 ...) time_1 (value_1 ...))
   ,(fold-right (lambda (var.value venv-term)
                  (let ((var (car var.value))
                        (val (cadr var.value)))
                    (term (venv-update ,venv-term (LVar ,var) time_1 ,val))))
                (term venv_1)
                (zip (term (var-name_1 ...)) (term (value_1 ...))))))
```

The metafunction benv-update/params works analogously.

The semantic function $\mathcal{A}$ evaluates the arguments of a call. Its counterpart in the model is the metafunction arg-eval:

```
(define-multi-args-metafunction arg-eval
  cps-language

  ((benv venv store time (Literal number_1))
   (Int number_1))
  ((benv venv store time (Literal true))
   (Bool true))
  ((benv venv store time (Literal false))
   (Bool false))

  ((benv_1 venv store time_1 lam-cont_1)
   (Clos lam-cont_1 benv_1 time_1))
  ((benv_1 venv store time_1 lam-user_1)
   (Clos lam-user_1 benv_1 time_1))
  ((benv_1 venv store time_1 lam-jump_1)
   (Clos lam-jump_1 benv_1 time_1))

  ((benv_1 venv_1 store time var_1)
   (venv-lookup venv_1 var_1 (benv-lookup benv_1 var_1)))
  ((benv venv store_1 time (GVar var-name_1))
   (store-lookup store_1 (Ref (GLoc var-name_1) (Literal "top"))))

  ((benv_1 venv_1 store_1 time_1
          (Trivcall label_1 prim-lang-call_1 exp_1 ...))
   (eval-primop benv_1 venv_1 store_1 time_1
          (prim-lang-call_1 exp_1 ...))))
```

The last argument to arg-eval is the expression to evaluate. Arg-eval dispatches on the type of expression and evaluates literals and lambda expressions directly. Note that $\mathcal{A}$ evaluates literals directly: In the executable models both functions are fused together and the cases of $\mathcal{K}_{Lang}$ become cases of $\mathcal{A}$.

Arg-eval uses the auxiliary functions venv-lookup and store-lookup to find the corresponding value for a variable. Note that trivial calls are also

expressions evaluated by $\mathcal{A}$. The last rule deals with trivial calls and delegates the evaluation to `eval-primop` the counterpart of $\mathcal{P}_{Lang}$ in the model.

$\widehat{\mathcal{A}}$ evaluates arguments in the abstract semantics and has the following counterpart in the model:

```
(define-multi-args-metafunction aarg-eval
  cps-language

  ((abenv avenv astore atime (Literal number_1))
   (AValue (AInt number_1)))
  ((abenv avenv astore atime (Literal true))
   (AValue (ABool true)))
  ((abenv avenv astore atime (Literal false))
   (AValue (ABool false)))

  ((abenv_1 avenv_1 astore atime var_1)
   (avenv-lookup avenv_1 var_1 (abenv-lookup abenv_1 var_1)))
  ((abenv avenv astore_1 atime (GVar var-name_1))
   (astore-lookup astore_1 (ARef (GLoc var-name_1) (Literal "top"))))

  ((abenv_1 avenv astore atime_1 lam-cont_1)
   (AValue (AClos lam-cont_1 abenv_1 atime_1)))
  ((abenv-1 avenv astore atime_1 lam-user_1)
   (AValue (AClos lam-user_1 abenv_1 atime_1)))
  ((abenv_1 avenv astore atime_1 lam-jump_1)
   (AValue (AClos lam-jump_1 abenv_1 atime_1)))

  ((abenv_1 avenv_1 astore_1 atime_1
            (Trivcall label_1 prim-lang-call_1 exp_1 ...))
   (aeval-primop abenv_1 avenv_1 astore_1 atime_1
            (prim-lang-call_1 exp_1 ...))))
```

`Aarg-eval` works similiar to `arg-eval` but operates on abstract environments and returns abstract values. As its concrete counterpart `aarg-eval` also evaluates literals directly.

When evaluating arguments the semantics always evaluates all call arguments. The following metafunctions evaluate a vector of expressions:

```
(define-multi-args-metafunction eval-arg-list
  cps-language
  ((benv_1 venv_1 store_1 time_1 (exp_1 ...))
   ,(map (lambda (exp)
           (term (arg-eval benv_1 venv_1 store_1 time_1 ,exp)))
         (term (exp_1 ...)))))

(define-multi-args-metafunction aeval-arg-list
  cps-language
  ((abenv_1 avenv_1 astore_1 atime_1 (exp_1 ...))
   ,(map (lambda (exp)
           (term (aarg-eval abenv_1 avenv_1 astore_1 atime_1 ,exp)))
         (term (exp_1 ...)))))
```

The metafunction `avalue-join` is the implementation of $\sqcup$ for $\widehat{\mathbf{D}}$ in the model:

```
(define-multi-args-metafunction avalue-join
```

```
  cps-language
  (((AValue dvalue_1 ...) (AValue dvalue_2 ...))
   (AValue
    ,@(lset-union (lambda (dval-1 dval-2)
                    (and (term (dvalue-weaker<? ,dval-1 ,dval-2))
                         (term (dvalue-weaker<? ,dval-2 ,dval-1))))
                  (term (dvalue_1 ...))
                  (term (dvalue_2 ...))))))
```

The Scheme procedure `lset-union` joins two sets that are given in the form of lists and uses the given equivalence predicate to compare values from both sets. The equivalence predicate in the example above is constructed using the metafunction `dvalue-weaker<?` — the implementation of $\sqsubseteq$ for denotable values. `Dvalue-weaker<?` is defined as follows:

```
(define-multi-args-metafunction dvalue-weaker<?
  cps-language

  ((halt halt) #t)
  (((AClos any_1 abenv_1 atime_1) (AClos any_2 abenv_2 atime_2))
   ,(and (term (lambda=? any_1 any_2))
         (term (atime-weaker<? atime_1 atime_2))
         (term (abenv-weaker<? abenv_1 abenv_2))))

  (((AInt number_1) (AInt number_1)) #t)
  (((AInt bot) (AInt any)) #t)
  (((AInt any) (AInt top)) #t)

  (((ABool bot) (ABool any)) #t)
  (((ABool any) (ABool top)) #t)
  (((ABool true) (ABool true)) #t)
  (((ABool false) (ABool false)) #t)

  (((ARef (GLoc var-name_1) (Literal "top"))
    (ARef (GLoc var-name_1) (Literal "top")))
   #t)
  (((ARef (HLoc label_1) (Literal string_1))
    (ARef (HLoc label_1) (Literal string_1)))
   #t)

  ((any any) #f))
```

In this definition, the case that considers abstract closures is interesting: Here, `dvalue-weaker<?` uses the metafunction `atime-weaker<?` to compare two points in time and `abenv-weaker<?` to compare two binding environments. The model uses a 0CFA time abstraction by default and therefore `atime-weaker<?` is defined by:

```
(define-multi-args-metafunction atime-weaker<?
  cps-language
  (((ATime any) (ATime any)) #t))
```

Checking whether an abstract binding environment approximates another environment in the sense of $\sqsubseteq$ involves checking all entries:

```
(define-multi-args-metafunction abenv-weaker<?
```

```
  cps-language
  (((ABEnv abenv-entries_1 ...) (ABEnv abenv-entries_2 ...))
   ,(every (lambda (entry-term-1)
             (my-find (lambda (entry-term-2)
                        (term (abenv-entry-weaker<? ,entry-term-1
                                                    ,entry-term-2)))
                      (term (abenv-entries_2 ...))))
           (term (abenv-entries_1 ...)))))
```

The definition above relates to the following definition for $\sqsubseteq$ on binding environments:

$$\widehat{\beta} \sqsubseteq \widehat{\beta}' \quad \Leftrightarrow \quad \forall v \in \mathrm{dom}(\widehat{\beta}) : \; \widehat{\beta}(v) \sqsubseteq \widehat{\beta}'(v)$$

There are a few more metafunctions needed that implement $\sqsubseteq$ for abstract values, variable environments, stores, and states:

```
(define-multi-args-metafunction avalue-weaker<?
  cps-language
  (((AValue dvalue_1 ...) (AValue dvalue_2 ...))
   ,(every (lambda (dval-term-1)
             (my-find (lambda (dval-term-2)
                        (term (dvalue-weaker<? ,dval-term-1 ,dval-term-2)))
                      (term (dvalue_2 ...))))
           (term (dvalue_1 ...)))))

(define-multi-args-metafunction avenv-entry-weaker<?
  cps-language
  (((var-name_1 atime_1 avalue_1) (var-name_2 atime_2 avalue_2))
   ,(and (term (var=? var-name_1 var-name_2))
         (term (atime-weaker<? atime_1 atime_2))
         (term (avalue-weaker<? avalue_1 avalue_2)))))

(define-multi-args-metafunction avenv-weaker<?
  cps-language
  (((AVEnv avenv-entries_1 ...) (AVEnv avenv-entries_2 ...))
   ,(every (lambda (entry-term-1)
             (my-find (lambda (entry-term-2)
                        (term (avenv-entry-weaker<? ,entry-term-1
                                                    ,entry-term-2)))
                      (term (avenv-entries_2 ...))))
           (term (avenv-entries_1 ...)))))

(define-multi-args-metafunction state-weaker<?
  cps-language
  (((AApply id_1 id_2 avalue_1 (avalue_2 ...) (avalue_3 ...) avenv_1
            astore_1 atime_1)
    (AApply id_3 id_4 (avalue_5 ...) (avalue_6 ...) avenv_2 astore_2 atime_2))
   ,(or (= (term id_1) (term id_3))
        (and (term (avalue-weaker<? avalue_1 avalue_4))
             (term (atime-weaker<? atime_1 atime_2))
             (term (avalue-vector-weaker<? (avalue_2 ...) (avalue_5 ...)))
             (term (avalue-vector-weaker<? (avalue_3 ...) (avalue_6 ...)))
             (term (avenv-weaker<? avenv_1 avenv_2))
             (term (astore-weaker<? astore_1 astore_2)))))
  (((AEval id_1 id_2 call-user_1 abenv_1 avenv_1 astore_1 atime_1)
```

```
    (AEval id_3 id_4 call-user_2 abenv_2 avenv_2 astore_2 atime_2))
   ,(or (= (term id_1) (term id_3))
        (and (term (call=? call-user_1 call-user_2))
             (term (atime-weaker<? atime_1 atime_2))
             (term (abenv-weaker<? abenv_1 abenv_2))
             (term (avenv-weaker<? avenv_1 avenv_2))
             (term (astore-weaker<? astore_1 astore_2)))))
  (((AEval id_1 id_2 call-cont_1 abenv_1 avenv_1 astore_1 atime_1)
    (AEval id_3 id_4 call-cont_2 abenv_2 avenv_2 astore_2 atime_2))
   ,(or (= (term id_1) (term id_3))
        (and (term (call=? call-cont_1 call-cont_2))
             (term (atime-weaker<? atime_1 atime_2))
             (term (abenv-weaker<? abenv_1 abenv_2))
             (term (avenv-weaker<? avenv_1 avenv_2))
             (term (astore-weaker<? astore_1 astore_2)))))
  ((any any) #f))
```

The metafunction above uses the auxiliary metafunction `avalue-vector-weaker<?`, not shown here, to compare vectors of abstract values.

Finally, the model also defines metafunctions that implement the correctness relation $\mathcal{R}$. Consider the cases for integers and booleans first:

```
(define-multi-args-metafunction correctness-int
  cps-language
  (((Int number_1) (AInt number_1)) #t)
  (((Int any) (AInt top)) #t)
  ((any any) #f))

(define-multi-args-metafunction correctness-bool
  cps-language
  (((Bool true)  (ABool true))  #t)
  (((Bool false) (ABool false)) #t)
  (((Bool any) (ABool top)) #t)
  ((any any) #f))
```

The correctness relation for denotable values searches the abstract values for an abstract counterpart that is considered a correct abstraction in the sense of $\mathcal{R}$:

```
(define-multi-args-metafunction correctness-avalue
  cps-language
  ((proc_1 (AValue dvalue_1 ...))
   ,(any (lambda (dval)
           (term (correctness-proc proc_1 ,dval)))
         (term (dvalue_1 ...))))
  ((int_1 (AValue dvalue_1 ...))
   ,(any (lambda (dval)
           (term (correctness-int int_1 ,dval)))
         (term (dvalue_1 ...))))
  ((bool_1 (AValue dvalue_1 ...))
   ,(any (lambda (dval)
           (term (correctness-bool bool_1 ,dval)))
         (term (dvalue_1 ...))))
  ((ref_1 (AValue dvalue_1 ...))
   ,(any (lambda (dval)
           (term (correctness-ref ref_1 ,dval)))
```

```
        (term (dvalue_1 ...))))
  ((any any) #f))
```

The metafunction `correctness-venv` checks whether an abstract variable environment is a valid abstraction of a given concrete one. According to the specification of $\mathcal{R}$ every value in the concrete variable environment must have an abstract counterpart:

```
(define-multi-args-metafunction correctness-venv-entry
  cps-language
  (((var-name_1 time_1 value_1) (var-name_2 atime_1 avalue_1))
   ,(or (not (and (term (var=? var-name_1 var-name_2))
                  (term (correctness-time time_1 atime_1))))
        (term (correctness-avalue value_1 avalue_1))))
  ((any any) #f))

(define-multi-args-metafunction correctness-venv
  cps-language
  (((VEnv venv-entries_1 ...) (AVEnv avenv-entries_1 ...))
   ,(every (lambda (entry)
             (every (lambda (aentry)
                      (term (correctness-venv-entry ,entry ,aentry)))
                    (term (avenv-entries_1 ...))))
           (term (venv-entries_1 ...))))
  ((any any) #f))
```

There are a few more correctness metafunctions involved for checking binding environments, stores, vectors with call arguments. These metafunctions are straightforward to define and I therefore omit the presentation. With these metafunctions at hand, the correctness relation for states is defined as:

```
(define-multi-args-metafunction correctness-state
  cps-language
  (((Eval  id_1 id_2 any_1 benv_1 venv_1 store_1 time_1)
    (AEval id_3 id_4 any_2 abenv_1 avenv_1 astore_1 atime_1))
   ,(and (term (call=? any_1 any_2))
         (term (correctness-benv benv_1 abenv_1))
         (term (correctness-venv venv_1 avenv_1))
         (term (correctness-store store_1 astore_1))
         (term (correctness-time time_1 atime_1))))
  (((Apply id_1 id_2 proc_1 (value_1 ...) (value_2 ...)
           venv_1 store_1 time_1)
    (AApply id_3 id_4 (AValue aproc_1) (avalue_3 ...) (avalue_4 ...)
           avenv_1 astore_1 atime_1))
   ,(and (term (correctness-proc proc_1 aproc_1))
         (term (correctness-avalue-vector (value_1 ...) (avalue_3 ...)))
         (term (correctness-avalue-vector (value_2 ...) (avalue_4 ...)))
         (term (correctness-venv venv_1 avenv_1))
         (term (correctness-store store_1 astore_1))
         (term (correctness-time time_1 atime_1))))
  ((any any) #f))
```

5.2.3 Reduction relations

With the metafunctions in place everything is set up to formulate the actual reduction relations on terms. The following code shows the concrete state transition ⇀ from *apply* to *eval* states:

```
(define concrete-transition
  (reduction-relation
   cps-language
     (--> (Apply id_1 id_2 (Clos any_1 benv_1 time_1) any_2 any_3
                 venv_1 store_1 time_2)
          (Eval ,(next-id) id_1
                (lambda-body any_1)
                (benv-update/params benv_1 (lambda-params any_1) time_2)
                (venv-update/params venv_1 (lambda-params any_1) time_2
                                    ,(append (term any_2) (term any_3)))
                store_1 time_2)
          ApplyClos)
```

The Redex macro `reduction-relation` defines a new reduction relation as a list of rules. Each rule starts with `-->`, followed by a pattern to match (again, the pattern may include ellipses and pattern variables), and a result term. In the example above, the pattern matches any concrete *apply* state and produces a new *eval* state. The metafunction `next-id` produces a fresh state id. `Benv-update/params` and `venv-update/params` are also metafunctions that update the binding environment and variable environments for multiple variables at once. The reduction relation needs a case for every concrete transition rule specified in Section 2.3. The remaining transition rules from *eval* to *apply* states all work very similar and I therefore present only the counterpart of PCallEval as an example:

```
   (--> (Eval id_1 id_2 (Call label_1 prim-proc-call_1 exp_1 ...)
              benv_1 venv_1 store_1 time_1)
        ,(let* ((new-time (term (time-tick time_1)))
                (op-exp
                   (term (call-op-exp
                            (Call label_1 prim-proc-call_1 exp_1 ...))))
                (op (term (arg-eval benv_1 venv_1 ,new-time ,op-exp)))
                (cont-exp
                   (term (call-cont-exp
                            (Call label_1 prim-proc-call_1 exp_1 ...))))
                (cont (term (arg-eval benv_1 venv_1 ,new-time ,cont-exp)))
                (args-exps
                   (term (call-arg-exps
                            (Call label_1 prim-proc-call_1 exp_1 ...))))
                (args (term (eval-arg-list benv_1 venv_1 store_1
                                           ,new-time ,args-exps))))
           (term (Apply ,(next-id) id_1 ,op (,cont) ,args venv_1
                                           store_1 ,new-time)))
        PCallEval)
        ...
```

This reduction rule uses metafunctions shown above to decompose a call into the expressions in operator, continuation, and argument position. To evaluate the expressions it uses the argument evaluation functions `arg-eval` and `eval-arg-list` also shown above.

The executable semantic model contains a second reduction relation that models the transition relation $\widehat{\Rightarrow}$ of the abstract semantics. Again, I show the transition rule from *apply* to *eval* states and one example of a rule from *eval* to *apply* states. Here is the counterpart of $\widehat{\text{ApplyClos}}$:

```
(define abstract-transition
  (reduction-relation
   cps-language

   (--> (AApply id_1 id_2 (AValue (AClos any_1 abenv_1 atime_1))
                (avalue_1 ...) (avalue_2 ...)
                avenv_1 astore_1 atime_2)
        ,(let* ((call   (term (lambda-body any_1)))
                (params (term (lambda-params any_1)))
                (new-benv (term (abenv-update/params
                                   abenv_1 ,params atime_2)))
                (all-args (append (term (avalue_1 ...))
                                  (term (avalue_2 ...))))
                (new-venv (term (avenv-update/params
                                   avenv_1 ,params atime_2 ,all-args))))
           (term ((AEval ,(next-id) id_1 ,call ,new-benv ,new-venv
                         astore_1 atime_2))))
        AbsApplyClos)
```

Note that resulting *eval* state is wrapped in an additional pair of parentheses: The reduction relation for abstract states returns a sequence of terms, each term denoting a state. The implementation of $\widehat{\text{PCallEval}}$ shows this clearly:

```
  (--> (AEval id_1 id_2 (Call label_1 prim-proc-call_1 exp_1 ...)
              abenv_1 avenv_1 astore_1 atime_1)
       ,(let* ((new-time  (term (atime-tick atime_1)))
               (args-exps
                  (term (call-arg-exps
                           (Call label_1 prim-proc-call_1 exp_1 ...))))
               (op-exps
                  (term (call-op-exp
                           (Call label_1 prim-proc-call_1 exp_1 ...))))
               (cont-exp
                  (term (call-cont-exp
                           (Call label_1 prim-proc-call_1 exp_1 ...))))
               (args
                  (term (aeval-arg-lost abenv_1 avenv_1 ,new-time
                                        ,args-exps)))
               (proc
                  (term (aarg-eval abenv_1 avenv_1 ,new-time ,op-exps)))
               (cont
                  (term (aarg-eval abenv_1 avenv_1 ,new-time ,cont-exp))))
          (term ,(map (lambda (p)
                        (term (AApply ,(next-id) id_1 (AValue ,p)
                                      (,cont) ,args
                                      avenv_1 astore_1 ,new-time)))
                      (term (filter-aprocs ,proc)))))
       AbsPCallEval)
```

This rule decomposes the call and evaluates the expressions in continuation, operator, and argument position. The result of the evaluation of the operator

expression is an abstract value. The metafunction `filter-aprocs` filters all procedure values — abstract closures or the `halt` continuation. The reduction rule then loops over the procedure values and creates an *apply* state for each value.

5.2.4 Generating traces

Redex provides a function called `traces` that computes reduction steps: Given an initial state and an reduction relation, `traces` finds all terms and displays an interactive graph showing how the reduction relation proceeds.

Figure 5.3 shows a screenshot of the state transition graph computed by `traces`. The transition uses the reduction rule for the concrete semantics and the following initial state (see Figure 5.2 for the source code):

```
(define ps-factorial-initial-state
  (term (Apply 1 0 (Clos ,ps-factorial-main22 (BEnv) (Time 0))
               (halt) ()
               (VEnv) (Store) (Time 0))))
```

Computing the trace for the abstract semantics is more complex. First, the abstract transition relation $\widehat{\Rightarrow}$ may produce more than one successor state. Second, the termination check poses another challenge: The new states must be checked against the set of visited states $\widehat{\mathcal{V}}$. The flow analysis only considers states not already included in $\widehat{\mathcal{V}}$ (in the sense of $\sqsubseteq$) as candidates for further state transitions.

Consequently, `traces` is not sufficient for computing the abstract state transition. The following procedure computes the flow analysis and exactly works as the procedure `analyze` shown in Figure 4.7:

```
(define (flow-analyze initial-state)
  (let ((unvisited (make-queue)))
    (enqueue! unvisited initial-state)
    (let lp ((visited '()))
         (if (queue-empty? unvisited)
             (reverse visited)
           (let* ((s (dequeue! unvisited))
                  (new-visited (cons s visited))
                  (successors (apply-reduction-relation
                                abstract-transition s)))
             (for-each
              (lambda (ns)
                (if (not (or (term (find-in-set
                                    ,ns ,new-visited))
                             (term (find-in-set
                                    ,ns ,(queue->list unvisited)))))
                    (enqueue! unvisited ns)))
              (if (null? successors) '() (car successors)))
             (lp new-visited))))))
```

The procedure `flow-analyze` applies the basic Redex function `apply-reduction-relation` to compute a single transition step for $\widehat{\Rightarrow}$ (specified by `abstract-transition`). This reduction yields a term that contains multiple states. `Flow-analyze` splits the terms and checks whether the resulting states have been already visited by calling the metafunction `find-in-set`. Only unvisited states

PLT Redex Reduction Graph

```
(Eval
 8
 7
 (Call
  321
  jump
  (LVar lp27)
  (Trivcall
   322
   +
   (Literal -1)
   (LVar n29))
  (Trivcall
   323
   *
   (LVar n29)
   (LVar r30)))
 (BEnv
  (c21 (Time 0))
  (lp27 (Time 1))
  (r30 (Time 2))
  (n29 (Time 2)))
 (VEnv
  (c21 (Time 0) halt)
  (lp27
   (Time 1)
   (Clos
    (Lambda-j
      31
      (n29 r30)
      (Call
       310
       test
       (Lambda-c
        33
        ()
        (Call
         331
         unknown-return
         (LVar c21)
         (LVar r30)))
       (Lambda-c
        32
        ()
        (Call
         321
         jump
         (LVar lp27)
         (Trivcall
          322
          +
          (Literal -1)
          (LVar n29))
         (Trivcall
          323
          *
          (LVar n29)
          (LVar r30))))
       (Trivcall
        311
        <
        (LVar n29)
        (Literal 2))))
    (BEnv
     (c21 (Time 0))
     (lp27 (Time 1)))
    (Time 1)))
  (r30 (Time 2) (Int 1))
  (n29 (Time 2) (Int 5)))
```

```
(Apply
 9
 8
 (Clos
  (Lambda-j
   31
   (n29 r30)
   (Call
    310
    test
    (Lambda-c
     33
     ()
     (Call
      331
      unknown-return
      (LVar c21)
      (LVar r30)))
    (Lambda-c
     32
     ()
     (Call
      321
      jump
      (LVar lp27)
      (Trivcall
       322
       +
       (Literal -1)
       (LVar n29))
      (Trivcall
       323
       *
       (LVar n29)
       (LVar r30))))
    (Trivcall
     311
     <
     (LVar n29)
     (Literal 2))))
  (BEnv
   (c21 (Time 0))
   (lp27 (Time 1)))
  (Time 1))
 ()
 ((Int 4) (Int 5))
 (VEnv
  (c21 (Time 0) halt)
  (lp27
   (Time 1)
   (Clos
    (Lambda-j
     31
     (n29 r30)
     (Call
      310
      test
      (Lambda-c
       33
       ()
       (Call
        331
        unknown-return
        (LVar c21)
        (LVar r30)))
      (Lambda-c
       32
       ()
       (Call
```

```
(Eval
 10
 9
 (Call
  310
  test
  (Lambda-c
   33
   ()
   (Call
    331
    unknown-return
    (LVar c21)
    (LVar r30)))
  (Lambda-c
   32
   ()
   (Call
    321
    jump
    (LVar lp27)
    (Trivcall
     322
     +
     (Literal -1)
     (LVar n29))
    (Trivcall
     323
     *
     (LVar n29)
     (LVar r30))))
  (Trivcall
   311
   <
   (LVar n29)
   (Literal 2)))
 (BEnv
  (c21 (Time 0))
  (lp27 (Time 1))
  (r30 (Time 4))
  (n29 (Time 4)))
 (VEnv
  (c21 (Time 0) halt)
  (lp27
   (Time 1)
   (Clos
    (Lambda-j
     31
     (n29 r30)
     (Call
      310
      test
      (Lambda-c
       33
       ()
       (Call
        331
        unknown-return
        (LVar c21)
        (LVar r30)))
      (Lambda-c
       32
       ()
       (Call
        321
        jump
        (LVar lp27)
        (Trivcall
         322
```

ApplyClos — CCallEval — ApplyClos

Font Size

11

Reduce   found 21 terms(possibly more to find)

Figure 5.3: Screenshot of PLT Redex showing a trace

go into the queue `unvisited`. The return value of `flow-analyze` is a Scheme list that contains all terms visited during the analysis. The definition of `find-in-set` is as follows:

```
(define-multi-args-metafunction find-in-set
  cps-language
  ((astate_1 (astate_2 ...))
   ,(my-find (lambda (s)
               (term (state-weaker<? ,s astate_1)))
             (term (astate_2 ...)))))
```

Given a state `astate_1` and sequence of states `(astate_2 ...)` the above function checks if the sequence already contains a state that approximates `astate_1`.

Having computed the visited sets $\mathcal{V}$ and $\widehat{\mathcal{V}}$, the implementation may check the correctness of the analysis. The Scheme procedure `check-correctness` is an implementation of Theorem 4.3 and uses `correctness-state` (the implementation of $\mathcal{R}_{State}$) to compare the concrete and abstract states:

```
(define (check-correctness real-visited flow-visited)
  (every
   (lambda (cstate)
     (printf "~%Considering concrete state:~%~a" cstate)
     (any (lambda (astate)
            (let ((abstracts?
                    (term (correctness-state ,cstate ,astate))))
              (printf "  ~a for ~a~%" abstracts? astate)
              abstracts?))
          flow-visited))
   real-visited))
```

The trace viewer of PLT Redex is designed to show a single trace of one reduction relation. In this case, however, there are two traces to show which are connected by the correctness relation. Hence, the standard `trace` can not display the results. The code for presenting the graph in Redex is tightly coupled to applying the reduction relation. Thus, it seemed more promising to use an external graph viewer to inspect the graphs. For this reason, I added functionality to export the terms as a GML file depicting the graphs and the correctness relation. Each state becomes a node with the complete term as its label. The numeric ids of the states define the edges between nodes and the correctness relation adds edges between concrete and abstract states. The traversal of the visited set is implemented as a simple reduction relation itself that generates a hash table that encodes the graph. To layout and inspect the graph I used the *yEd* graph drawing software [*Wiese et al.*, 2001].

Chapter 6

Implementation

Polyvariant control-flow analyses for programs of realistic sizes are expensive to compute. A straightforward implementation of the flow-analysis semantics as presented in the preceding chapters is almost unusable when applied to input programs of more than 1,000 lines of code: Scalability is the main challenge. As part of this work, I developed a sophisticated implementation of the analysis. The implementation is fast enough to analyze programs with thousands of lines in a few minutes. The implementation is about 15,000 lines of Scheme code. It handles the complete CPS intermediate language of the transformational compiler and the primops that are specific to Scheme and PreScheme.

This chapter gives an overview of the implementation and discusses implementation techniques used to improve the performance of the analysis.

6.1 Abstract syntax

Before turning to the implementation of the semantic domains of the domains a closer look at the representation of the abstract syntax tree is necessary. The transformational compiler uses a regular term structure geared towards efficiency. There are five types of nodes: literals, variables, variable references, lambda terms, and calls. Operations on the abstract syntax tree are carried out using side effects and destructive updates. Moreover, the program representation has the following properties:

- Each binding in the program has a unique variable name.
- Each variable and lambda node has an unique integer that may be used as a hash key.
- Every node has a pointer to its parent node (the parent of a call is a lambda node unless the call is trivial). This allows navigating the syntax tree quickly.
- Each variable node maintains a list of pointers to its references in the program.
- Each variable node contains a field pointing to its binder.

This representation enables substantial performance increases in a compiler [*Kennedy*, 2007].

The semantics assigns an unique label τ to each term, that allows comparing terms intensionally. In the context of the implementation assigning labels is superfluous: Each node is an instance of a record and comparing nodes intensionally is via `eq?`.

6.2 Semantic domains

For the analysis, a memory efficient representation is almost as important as a time efficient representation. Even for small programs the set of visited states may grow quickly to a few thousand states and require large amounts of memory (see Section 6.5).

This section identifies which operations occur frequently on which domains and discusses representations for these domains such that the common operations run fast.

6.2.1 Representing time

By default the implementation uses a 1CFA time set (see Section 4.5). Recall how the semantics advances time: Only transitions from *eval* states advance the time with $\widehat{tick}$. Transitions from *apply* states just pass the current time to their successor state. Since the 1CFA point in time is defined as the last dynamic call that occurred, $\widehat{tick}$ goes back to returning the call being subject of the *eval* state. Thus, 1CFA points in time are of a distinct record type that contains a pointer to the call node in the syntax tree:

```
(define-record-type time-1cfa :time-1cfa
  (really-make-time call-site uid)
  time?
  (call-site time-call-site)
  (uid time-uid))
```

The additional `uid` field exists to ease debugging: A smart constructor adds an unique number to each time record. The external representation of a time record prints this number instead of the call node because this is significantly shorter and easier to read. Computing $\sqsubseteq$ for time goes back to check the intensional equality of the wrapped call nodes.

Representing the 0CFA time set is trivial because $\widehat{\mathbf{Time}}$ is a singleton set in this case. Thus, a distinct record type with exactly one instance suffices.

6.2.2 Representing the binding environment

Consider how binding environments in the executable model of the semantics work: The binding environment maps variables to points in time. Each transition from an *apply* state augments a binding environment by one or more new entries. Adding an entry yields a new binding environment that — besides the new entry — shares all entries with the original binding environment. Binding environments are created whenever the $\widehat{\mathcal{A}}$ evaluates a lambda expression.

That is, the number of binding environments created is linked to the number of distinct closures and may be large.

Searching the binding environment to find the binding time of a variable occurs frequently because it is a part of the evaluation of a variable. Typically, binding environments do not have too many entries: Each application adds one entry for each parameter of a lambda expression. During my experiments, I rarely encountered programs where the number of entries in a binding environment exceeded one hundred entries.

A further consideration concerns the approximation relation $\sqsubseteq$ for binding environments (see Section 4.1). During the termination check the flow analysis compares a state with the states in the visited set using $\sqsubseteq$. This, in turn, may involve checking the approximation relation for binding environments: *Eval* states contain a binding environment, and $\sqsubseteq$ on sum lattices works summandwise. Comparisons of *apply* states also imply comparing binding environments if the *apply* states contain abstract closures. Computing $\sqsubseteq$ for two binding environments is costly: This means traversing all elements in the domain of one environment and comparing the images from both environments using $\sqsubseteq$.

I have experimented with three implementations for binding environments based on tables, search trees, and association lists. While hashtables provide a fast access the model is intrinsically stateful. Extending a binding environment for one new entry goes back to copying the complete hashtable and updating a single entry in the copy — which is very space inefficient. Search trees have similar problems. Here, a very simple representation turned out to be the most successful: sorted association lists.

The association list that holds the environment entries resides in a record that represents the binding environment. The list is sorted by the unique id assigned to each variable (see Section 6.1). This allows the procedure that implements $\sqsubseteq$ to step linearly through both lists simultaneously and compare every point in time for each variable. Comparison of binding environment therefore has in linear complexity.

6.2.3 Representing variable environments

The interactions between the garbage collector of the analysis and the variable environment require special attention. Thus, I will discuss the representation of the variable environment in conjunction with the garbage collector (see Section 6.4.3).

6.2.4 Representing abstract values

The domain for denotable values $\widehat{\mathbf{D}}$ is a powerset over the sum of each abstract value type. An abstract value type either belongs to a core value type — procedure values from $\widehat{\mathbf{Proc}}$ or references from $\widehat{\mathbf{Ref}}$ — or to a language-specific value type ($\widehat{\mathbf{Bas}}_{Lang}$ and $\widehat{\mathbf{Comp}}_{Lang}$). This section describes the interface of the analysis that enables the user to introduce new abstract value types. However, before considering language-specific value domains, I will discuss the core value types: procedure values and references.

A procedure value from $\widehat{\mathbf{Proc}}$ is either an abstract closure or the `halt` continuation. Both domains are easy to represent using records. The `halt` contin-

uation gets a new record type without fields for which only one instance exists. Closure records have a field for each element of the triple (a lambda node, a binding environment, and a point in time).

Abstract references consist of a literal node and an abstract location (see Section 4.1). The implementation wraps both constituents into a new record type. Abstract locations, in turn, also have their own record type. This allows distinguishing heap locations from global variable locations.

None of the core transition rules needs specific knowledge of the language-specific abstract value types; the rules merely pass values from caller to callee. However, this does not apply to the evaluation of literal nodes, the termination check, and the garbage collector:

- The evaluation function $\widehat{\mathcal{K}}_{Lang}$ recognizes literal nodes and evaluates these literals.
- The termination check compares a possibly new state with the states from the visited set. These states include language-specific abstract values in their argument vectors and variable environments. Thus, it is necessary to compute the approximation relation $\sqsubseteq$ for all abstract value types.
- Abstract values may contain references to other values. A garbage collector (see Section 6.4) that traces reachable values must know how to trace language-specific abstract values.

To define a new abstract value domain the user provides procedures that implement the functionality mentioned above. Such a definition consists of the following parts:

- A symbol, which serves as an unique type name.
- A procedure implementing $\sqsubseteq$ for this value domain.
- A type predicate that recognizes values of the new domain.
- A constructor that primops call to generate a new language-specific value.
- A procedure for evaluating literals (if the value type has literals).
- A so-called *GC follow* procedure that returns the references of an abstract value.

Here are two examples for such definitions:

Example #1: Boolean values Figure 6.1 shows the implementation of the Scheme boolean value type. The domain for boolean values consists of the elements $\{\bot_{Bool}, \texttt{true}, \texttt{false}, \top_{Bool}\}$. In practice, bottom is not necessary. Hence, the implementation only creates three records using `really-make-basic-value` and binds them to `the-scheme-true`, `the-scheme-false`, and `the-scheme-boolean-top`. The code uses the records of type *basic value* as a common representation. The `type` field of a basic value record contains the type name (`boolean` in this case) and distinguishes value types. A common representation simplifies printing and inspecting basic values.

`Make-scheme-boolean` is the constructor for boolean values that primops use to create abstract booleans. Given a Scheme boolean the procedure returns

```
(define the-scheme-true
  (really-make-basic-value 'boolean #t))

(define the-scheme-false
  (really-make-basic-value 'boolean #f))

(define the-scheme-boolean-top
  (really-make-basic-value 'boolean 'top))

(define (make-scheme-boolean value . ignore)
  (cond
    ((eq? #t value)     the-scheme-true)
    ((eq? #f value)     the-scheme-false)
    (else               the-scheme-boolean-top)))

(define (scheme-boolean<? bool-1 bool-2)
  (or (eq? bool-2 the-scheme-boolean-top)
      (and (eq? bool-1 the-scheme-true)
           (eq? bool-2 the-scheme-true))
      (and (eq? bool-1 the-scheme-false)
           (eq? bool-2 the-scheme-false))))

(define-absval-type boolean
  make-scheme-boolean
  value-from-literal
  (make-type-predicate-proc 'boolean)
  scheme-boolean<?
  #f)
```

Figure 6.1: Defining an abstract value type for boolean values

the corresponding element of the abstract domain. Primops use `make-scheme-boolean` also to construct the top value of the domain — the value that represents true and false. `Scheme-boolean<?` is the implementation of $\sqsubseteq$ on values of $\widehat{\mathbf{Boolean}}$.

`Define-absval-type` registers the new value type under the name `boolean` with the analysis. Internally, the analysis stores the given procedures under that name in a table. The procedure `value-from-literal` (not shown in the figure) mentioned here is the evaluation function for boolean literals ($\widehat{\mathcal{K}}_{Lang}$ in the semantics): Given a literal node it checks whether the node is a boolean literal and calls `make-scheme-boolean` to create a value if applicable. The last argument of `define-absval-type` indicates whether abstract values of this domain may contain references to other values. (For booleans this is not the case.)

Computing $\sqsubseteq$ With the definitions of the approximation relation for language-specific value domains, the implementation composes $\sqsubseteq$ for all summands of $\widehat{\mathbf{D}}$. Given two sets $\widehat{d_1}$ and $\widehat{d_2}$ from $\widehat{\mathbf{D}}$ the procedure `denotable-value<?` computes

$\widehat{d}_1 \sqsubseteq \widehat{d}_2$ according to the specification shown in Section 4.1: The procedure checks whether each element of $\widehat{d}_1$ has a counterpart in $\widehat{d}_2$ such that $\sqsubseteq$ holds for the elements. Thus, $\sqsubseteq$ on $\widehat{\mathbf{D}}$ goes back to computing the approximation relation on the summands of $\widehat{\mathbf{D}}$, implemented as `single-denotable-value<?`:

```
(define (single-denotable-value<? v-1 v-2)
  (cond
   ((and (proc? v-1) (proc? v-2))
    (proc<? v-1 v-2))
   ((and (user-defined-abs-value? v-1)
         (user-defined-abs-value? v-2))
    (abs-value<? v-1 v-2))
   ((and (abs-reference? v-1) (abs-reference? v-2))
    (abs-reference<? v-1 v-2))
   ((and (abs-compound? v-1) (abs-compound? v-2))
    (abs-compound<? v-1 v-2))
   (else #f)))
```

This procedure checks whether the given values belong to the core value types (procedure values or references), abstract compound values (discussed in the next paragraph), or one of the language-specific value domains and delegates the check to the corresponding implementation of $\sqsubseteq$. `User-defined-abs-value?` identifies language-specific values by applying all registered type predicates to the value. The procedure `abs-value<?` calls the user defined implementation of $\sqsubseteq$ for the value type in question. The predicate `abs-compound?` identifies compound values and delegates the check to the corresponding comparison procedure `abs-compound<?`, which, in turn, may delegate the check further to an implementation provided by the user.

Example #2: Cells A cell is a heap object and its abstract counterpart therefore is a reference to an object in the abstract store. That is, cells belong to the domain of abstract compound values $\widehat{\mathbf{Comp}}_{Lang}$. This section describes the implementation of cells within the analysis as an example for such value domains.

Figure 6.2 shows the code that defines the value type for cells. The code uses *abstract compound* records. Abstract compounds are a common representation of compound values offered by the analysis framework to simplify and unify the implementation. For explanatory reasons the example code uses abstract compounds but does not use the common procedures offered by the analysis.

`Make-scheme-cell` creates a new instance of an abstract compound value that simulates a Scheme cell. The argument `cell-contents` is an abstract value that denotes the initial contents of the cell. To modify the store according to the specification of stob creation (see Figure 4.12) `make-scheme-cell` uses the auxiliary procedure `allocate-stob`. `Allocate-stob` carries out a series of update operations on the given store:

- Create (and return) an abstract reference $\widehat{ref}_{stob}$ of the call node `id`, the time `time`, and the selector name `stob`.
- Create an abstract reference $\widehat{ref}_{content}$ from `id`, `time`, and the selector name `cell-contents`.

```
(define cell-type-name (enumerand->name (enum stob cell) stob))

(define (make-scheme-cell store id time cell-contents)
  (allocate-stob store id time
                 cell-type-name (no-sub-type)
                 (list (cons 'content cell-contents))))

(define (scheme-cell? thing)
  (and (abs-compound? thing)
       (eq? cell-type-name
            (abs-compound-discriminator thing))))

(define (scheme-cell-ref store refs)
  (apply union-dvalues
         (map (lambda (ref)
                (store-lookup store (make-abs-reference
                                     (abs-reference-location ref)
                                     'content)))
              refs)))

(define (scheme-cell-set store ref new-val)
  (store-update store (make-abs-reference
                       (abs-reference-location ref)
                       'content)
                new-val))

(define (scheme-cell<? c-1 c-2)
  (let ((s-1 (cdr (assq 'store (abs-compound-values c-1))))
        (r-1 (cdr (assq 'content (abs-compound-values c-1))))
        (s-2 (cdr (assq 'store (abs-compound-values c-2))))
        (r-2 (cdr (assq 'content (abs-compound-values c-2)))))
    (denotable-value<? (store-lookup s-1 r-1)
                       (store-lookup s-2 r-2))))

(define-absval-compound-type cell-type-name
  make-scheme-cell
  (lambda ignore #f)
  scheme-cell?
  scheme-cell<?
  scheme-cell-gc-follow)
```

Figure 6.2: Defining an abstract value type for cells

- Extend the store with the reference $\widehat{ref}_{content}$ and the value `cell-contents`.
- Create a fresh abstract compound record with the following contents: The type field is filled with `cell-type-name` to indicate the stob type. An association list that consists of the valid selector names and the references to the store entries that hold the values for these stob fields.
- Finally, `allocate-stob` extends the store with the reference $\widehat{ref}_{stob}$ and the abstract compound value and returns this reference.

Adding an abstract compound record `allocate-stob` breaks the invariant that the store contains only denotable values. As mentioned in Section 4.4.2 this happens on purpose: The information stored in an abstract compound is very useful for implementing the garbage collector as will become clear in Section 6.4.

The procedure `scheme-cell-ref` returns the value stored in a cell. Technically, arbitrary types of abstract values may occur as arguments to `cell-ref`. `Scheme-cell-ref` therefore narrows down the arguments to a set of references to abstract cells. With the described representation this is simple: First identify all abstract references in the set and among those find references to cells — this, in turn, may be accomplished by looking at the reference that consists of the location of the original reference and the `stob` selector. If the store returns an abstract compound that represents a cell for this reference, the reference will be considered. `Scheme-cell-ref` searches the store with these references and retrieves a number of denotable values that `union-dvalues` merges into a single value.

The code uses the macro `define-absval-compound-type` to register the new value domain with the analysis. Besides the procedures just described, this macro call also includes the GC follow procedure for cells bound to `scheme-cell-gc-follow`. This procedure merely extracts the reference used to store the content of a cell from an abstract compound record. Since cells do not have a literal form the definition passes a procedure that returns `#f` instead of a real implementation.

6.2.5 Implementing the visited set

An efficient implementation of the visited set is critically important for the efficiency of the analysis in total. Recall how the analysis proceeds for executing a single state transition (see Figure 4.7): Starting from a state the analysis computes the successor states. For each of the successor states the analysis now carries out the termination check: It checks whether it visited this state — or a state that approximates this state — before. Only if this is not the case, the state will be considered for further state transitions. Hence, the termination checks need to compare a state against the states in the visited set using the approximation relation $\sqsubseteq$. The implementation of the visited set addresses this task using the following techniques:

- *Separate visited sets* According to the specification of $\sqsubseteq$ only *apply* states may approximate *apply* states and only *eval* states approximate *eval* states. Hence, the implementation keeps *apply* and *eval* states in separate sets.
- *Order of checking* Checking $\sqsubseteq$ for state tuples works component-wise: If all components of a state tuple are related by $\sqsubseteq$, the approximation relation

holds for the whole state. Therefore, it makes sense to order the checks for the components such that the checks for components that are cheap happen first and expensive checks are carried out later.

- *Indexed sets* The subset of states to compare the new state with should be as small as possible and include only candidates that have a high chance of being related by $\sqsubseteq$. An index to find these states quickly is necessary.
- *Cache expensive checks* Some computations of $\sqsubseteq$ are intrinsically expensive. As mentioned in Section 6.2.2, binding environments qualify for this category. To avoid the re-computation a cache for the results is necessary.

Consider the idea of sets with an index first. Both sorts of states have at least one component for which the approximation relation is cheap to compute. This component serves as an index. For *eval* states this is the call node component. Nodes in the syntax tree are compared by their label and this, in turn, works by pointer comparison.

Indexing *apply* states is a more complicated. Here, the component that contains the procedure value is the index: A procedure values is either the `halt` continuation or an abstract closure. For *apply* states the lambda node of the abstract closure serves as the index. (A special case is necessary to index applications of the `halt` continuation.) Note that using the lambda node as the index also justifies the decision to have exactly one procedure value per *apply* state. Multiple procedure values — especially abstract closures over distinct lambda terms — would seriously complicate indexing *apply* states.

Eq hashtables Visited sets are implemented as hashtables. As discussed, for both sorts of states a node in the abstract syntax tree serves as the index. That is, a hash function for nodes in the syntax tree is necessary. *Eq hashtables* use the memory address of the object as a basis for computing the hash value. For the purpose aspired, eq hashtables would be optimal because they allow finding the relevant states in a large visited set very quickly while requiring almost no effort for computing the hash function.

Unfortunately, Scheme 48 did not have a eq hashtables at its disposal. Hence, in preparation for an efficient visited set I extended the Scheme 48 virtual machine with support for eq hashtables (called *id tables* in the implementation).[1]

The flow analysis algorithm in Figure 4.7 also checks whether the successor state has an approximation in the queue of states not visited yet — called the *state queue*. This queue may be very long (see Section 6.5). Therefore, state queues also use the implementation techniques as the visited set. In the implementation, state queues are a specialized form of visited sets with additional procedures to enqueue and dequeue states.

These considerations and preparations are the basis of my implementation of the visited set. In summary, the visited set is a record with fields that hold id tables for *eval* and *apply* states. The tables use the call node of an *eval* state and the lambda node of an *apply* state as the index, respectively.

[1] Eq hashtable require support from the garbage collector. By default Scheme 48 uses a two-space garbage collector that copies live values from the old heap space to new heap space. The id-table mechanism rehashes all entries of all tables after each garbage collection to reflect the new addresses. Recent research shows implementation techniques that also work in connection with a generational garbage collector and promise a better performance [*Ghuloum and Dybvig*, 2007].

6.3 Description of flow information

Section 4.4.2 briefly introduced the abstraction function $\widehat{\mathcal{S}}$ that returns the abstract counterpart of a given Scheme value. This abstraction function is necessary because initially Scheme programs are not passed in the form of source code to the analysis: The user passes a Scheme procedure and then starts the analysis. The process of creating abstract values with $\widehat{\mathcal{S}}$ is as follows:

- For procedure values $\widehat{\mathcal{S}}$ uses the byte-code optimizer to convert the byte code of the procedure to CPS nodes. That is, $\widehat{\mathcal{S}}$ reconstructs the source code.
- For other value types $\widehat{\mathcal{S}}$ returns the abstract counterpart directly without creating source code first.

Note that $\widehat{\mathcal{S}}$ operates on the transitive closure of all values referenced in the procedure passed to the analysis. This, however, causes serious problems in practice. Consider the following Scheme procedure:

```
(define (foo n)
  (if (even? n)
      42
      (error "Error: n is not even" n)))
```

This procedure either returns 42 or uses the Scheme function `error` to raise an error and abort the program. The program seems to be simple and the analysis should be simple too.

Consider how $\widehat{\mathcal{S}}$ proceeds on `foo`: It finds the reference to `error` and converts it to CPS source code. `Error`, however, is part of a complex exception system, so the pre-pass must consider this system too: This involves the abstraction of record types that represent exceptions and more procedures. The exception system, in turn, is implemented using `call-with-current-continuation` and the pre-pass must consider this procedure. Now, the code involved gets even more complex because `call-with-current-continuation` contains references to many complicated procedures of the Scheme 48 run-time system and even the debugger.

Thus, an analysis of the small procedure `foo` would involve large parts of the run-time system and become very complex. Most of the flow information computed, however, is not relevant as the run-time system and the standard libraries are not subject to an optimization.

One solution to this problem considers *modular* or *componential* flow analyses as used by Flanagan to implement a static debugger for Scheme code [*Flanagan*, 1997; *Flanagan and Felleisen*, 1999]. This approach formulates the flow analysis in form of a constraint system that uses simple equations. The equations are derived from the source code and a compression algorithm reduces the size of the constraint system. Thus, it is possible to pre-compute a compressed constraint system. Analyzing a program then consists of deriving the equations for the programs and loading the pre-computed set of equations. However, the equation system still must be solved altogether.

Flanagan's approach does not work with the analysis as specified in this dissertation for two reasons: First, the analysis is specified as an abstract interpretation. Second, the flow information for the library still needs to be derived

from the source code. An alternative approach considers contracts. Contracts work like assertions, however, have a clear formal underpinning in higher-order languages [*Findler and Felleisen*, 2002]. Recent research shows that contracts may serve as the source for flow information [*Meunier et al.*, 2006]. However, the connection between contracts and flow information is indirect: Contracts primarily exist to maintain assertions and not to describe flow information of libraries.

In the context of a partial evaluator for C called *Tempo* [*Consel et al.*, 2004] Consel et al. use a syntactic description for the behavior of external functions. If the source code of a C function is not available to the partial evaluator, for example because the function belongs to the standard library, the user may instead specify a piece of C code that behaves in the partial evaluation like the real code [*Consel et al.*, 1998].

The analysis specified in this dissertation uses a similar approach and offers the following techniques to specify the flow behavior of library code:

Replacement code $\widehat{\mathcal{S}}$ turns procedures into CPS code using the byte-code optimizer. The user, however, may define a procedure directly as CPS code and $\widehat{\mathcal{S}}$ uses this code instead of converted code. This makes is possible to define a replacement for a procedure for use in the analysis.

Flow primops Calls in the replacement code can use all primops supported by the analysis. Additionally, there is a special set of flow primops that directly create abstract values or modify the store.

The implementation described in this chapter uses the above techniques to describe the flow behavior of the most important Scheme library functions. The changes required for supporting this techniques in the implementation are as follows:

- Definitions of replacements reside in a table that maps procedure values to the corresponding replacement code.
- $\widehat{\mathcal{S}}$ must be changed to consult the table before converting a procedure to CPS code.
- The analysis needs an additional evaluation function that evaluates flow primops.

Examples Consider the following Scheme code that defines the replacement code for `call-with-current-continuation`:

```
(define vcall/cc
  (let ((k (make-cont-vvar 'k-callcc))
        (esc (make-user-vvar 'proc)))
    (make-user-vlambda
     (list k proc) #f (vlambda-proto matched)
     (make-vcall 'call
                 (vector (make-vref k)
                         (make-vref proc)
                         (make-vref k))))))
```

The Scheme functions `make-cont-vvar`, `make-user-vvar`, `make-vref`, `make-user-vlambda`, and `make-vcall` are simplified constructors for CPS nodes. Defining `call-with-current-continuation` in terms of CPS code is straightforward and therefore the replacement code consists only of a single user-defined procedure. The procedure has two arguments: A continuation argument with the name `k-callcc` and an argument `proc` for the procedure to call. `(vlambda-proto matched)` selects how the analysis matches the arguments. `Matched` indicates the standard protocol: The argument count must exactly match the parameter count.[2] The body of the procedure contains a call to `proc` with the current continuation `k` as the argument.

The second example is the replacement code for `error`:

```
(define verror
  (let ((x (make-user-vvar 'err-msg))
        (k (make-cont-vvar 'k-err)))
    (make-user-vlambda
     (list k x) #f (vlambda-proto raw)
     (make-vcall 'halt (vector (make-vref k) (make-vref x))))))
```

Again, the replacement code defines a user-defined procedure. The body of this procedure uses the flow primop `halt`. This flow primop directly controls the analysis and ensures that successor state is an *apply* state applying the halt continuation `halt`.

6.4 Analysis with Garbage Collection

The correctness relation for the flow analysis allows false positives. That is, an abstract value may include a representative for a concrete value that never occurs in any concrete program run.

This property can be the source of confusion. Consider how the variable environment works: Applications extend the variable environment with new values. Thereby the analysis merges the new argument values with values from previous applications. Thus, if a program contains multiple distinct calls to the same lambda expression the variable environment merges the values from all calls even though the calls are completely unrelated. A polyvariant time abstraction such as 1CFA overcomes this problem to some extent because it distinguishes calls by their context. However, polyvariance does not eliminate the problem completely since the number of points in time is still finite.

One important observation in this context concerns the lifetime of abstract entities. In contrast to values in a concrete program run abstract values have an unlimited lifetime. Might and Shivers were the first to describe this observation and propose the concept of garbage collection for abstract values [*Might and Shivers*, 2006]. That is, during a flow analysis a garbage collector checks which abstract values in the variable environment and the store are reachable through the state under consideration and removes unreachable values from the variable environment and the store.

Might and Shivers claim that the benefits of garbage collection are twofold: First, the flow analysis is more precise since it avoids merging unrelated infor-

[2] Replacement code may use `raw` to turn off argument matching completely. This is necessary to replace procedures with variable arity.

mation. Second, garbage collection may improve the performance of an analysis since the flow analysis deals with less abstract values and visits less states.

With these prospectives in mind I extended my analysis implementation to include a garbage collector extending and adapting Might's and Shivers's garbage collection algorithm. It turned out that writing a correct and efficient implementation of the garbage collection algorithm is one of the most challenging parts of the implementation. Early versions of the collector were very close to Might's and Shivers's definition. However, the analysis with the garbage collector actually ran significantly longer — for large programs with large amounts of abstract values the running time for garbage collection even dominated the total analysis time. This was especially the case for complex programs which yield more than 40,000 states (without garbage collection). Note that this effect is not noticeable with the small programs considered by Might and Shivers.

The following sections introduce a *global garbage collector*. This collector is a variation of Might's and Shivers's original collection algorithm (which I call the *local garbage collector*) that works well for large programs. The differences are as follows:

- The global collector affects the state under consideration and all states in the state queue while the local collector only considers a single state.
- The local collector restricts the variable environment of the successor state. The global collector uses a single variable environment and distinguishes versions of this environment using an efficient timestamp mechanism.

Note that the global collector is a method that primarily aims at improving the analysis speed — increasing the precision comes second. In particular, the global collection does not ensure a specific level of precision as the local collection. For example, Might and Shivers use the precision guarantees of the local collection to analyze the structure of binding environments [*Might and Shivers*, 2006]. Here, they exploit that the reachability of values in the analysis corresponds to the reachability of values in the concrete program run. In the global collector, the benefits to precision are almost unpredictable because the collector also takes the states from the queue into account. Hence, the set of live values in an analysis using the global collector does not correspond to the set of live values during a concrete program. In that sense, an analysis with global collection is less expressive. The global collector is, however, useful to improve the analysis time of larger programs.

Also the global collector makes it easy to borrow ideas from real garbage collectors that improve the running time of the collector significantly. In the preceding chapters and sections some preparations and design choices geared towards the garbage collector were already made. This section weaves this strands together and reports on implementation techniques for flow analysis garbage collectors.

6.4.1 Semantics with global garbage collection

The global garbage collector has the following mode of operation: First, the transition from *apply* to *eval* states checks whether a garbage collection is necessary and possibly invokes the collector. Garbage collection consists of two phases: Starting from a set of live objects, the *root set*, the collector traces all

$$\frac{\text{length}(\widehat{c}^*\S\widehat{d}^*) = n}{\widehat{\varsigma} = (\{((\lambda\ (p_1 \ldots p_n)\ call)_\tau, \widehat{\beta}, \widehat{t}_b)_\omega\}, \widehat{c}^*, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t})_{\omega'} \widehat{\Rightarrow} (call, \widehat{\beta}', \widehat{ve}', \widehat{\sigma_{gc}}, \widehat{t})_{\omega''}} \quad (\widehat{\text{ApplyClosGC}})$$

$$\text{where} \begin{cases} \widehat{\beta}' & = \widehat{\beta}[p_i \mapsto \widehat{t}] \\ (\widehat{ve_{gc}}, \widehat{\sigma_{gc}}) & = collect(\widehat{\varsigma} \sqcup \widehat{\mathcal{RS}}, \widehat{ve}, \widehat{\sigma}, \{(p_1, \widehat{t}), \ldots, (p_n, \widehat{t})\}, \varnothing) \\ \widehat{ve}' & = \widehat{ve_{gc}} \sqcup [(p_i, \widehat{t}) \mapsto (\widehat{c}^*\S\widehat{d}^*)_i] \end{cases}$$

Figure 6.3: Transition from *apply* states with garbage collection

bindings and references that are reachable. The second phase then restricts the variable environment and store to the reachable bindings and references. This section establishes a formal understanding of the collection mechanism.

A few preparatory changes to the semantics are necessary. To ease the discrimination of objects in the abstract semantics, that are of interest for the garbage collector, objects now include a so-called *GC id* that identifies them unambiguously:

$$\widehat{\mathbf{GCId}} = \mathbb{N}$$

The important property of GC ids is uniqueness ensured by the function *fresh-gcid*:

$$\textit{fresh-gcid} : \to \widehat{\mathbf{GCId}}$$

Each application of this function yields a fresh member of $\widehat{\mathbf{GCId}}$ that has never been returned by any application of *fresh-gcid* before.

The following objects in the semantics are of interest for the garbage collector: *eval* and *apply* states, closures, references, and compound values. These values are called *GC relevant*. The corresponding semantic domains now include an additional component for the GC id:

$$\begin{aligned} \widehat{\mathbf{Eval}} &= \texttt{Call} \times \widehat{\mathbf{BEnv}} \times \widehat{\mathbf{VEnv}} \times \widehat{\mathbf{Store}} \times \widehat{\mathbf{Time}} \times \widehat{\mathbf{GCId}} \\ \widehat{\mathbf{Apply}} &= \widehat{\mathbf{Proc}} \times \widehat{\mathbf{D}}^* \times \widehat{\mathbf{D}}^* \times \widehat{\mathbf{VEnv}} \times \widehat{\mathbf{Store}} \times \widehat{\mathbf{Time}} \times \widehat{\mathbf{GCId}} \\ \widehat{\mathbf{Clo}} &= \texttt{Lam} \times \widehat{\mathbf{BEnv}} \times \widehat{\mathbf{Time}} \times \widehat{\mathbf{GCId}} \end{aligned}$$

For states and abstract closures the GC id is written as a subscript ω instead of a tuple component. References do not require an id. Compound values such as vectors, records, and cells require a GC id.

Each transition rule assigns a new GC id to the resulting state using *fresh-gcid*. Since this change to the transition rules (see Chapter 4) is straightforward I do not list the modified rules. Adapting the semantic functions $\widehat{\mathcal{P}}_{Lang}$, $\widehat{\mathcal{K}}_{Lang}$, and $\widehat{\mathcal{A}}$ is straightforward: Each function now assigns a fresh GC id if it creates a new GC-relevant value. The argument evaluation function $\widehat{\mathcal{A}}$ serves as an example:

$$\widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,lam \quad = \quad \{(lam, \widehat{\beta}, \widehat{t})_\omega\} \qquad \text{where } \omega = \textit{fresh-gcid}$$

Adding GC ids completes the preparations for the garbage collector. In a programming language implementation a shortage of heap space triggers a garbage collection. The analysis performs a garbage collection when the analysis

$$\begin{aligned}\mathit{collect} : \mathcal{P}(\widehat{\textbf{State}}) \times \widehat{\textbf{VEnv}} \times \widehat{\textbf{Store}} \times \mathcal{P}(\texttt{Var} \times \widehat{\textbf{Time}}) \times \mathcal{P}(\widehat{\textbf{Ref}}) \rightarrow \\ \widehat{\textbf{VEnv}} \times \widehat{\textbf{Store}}\end{aligned}$$

$$\mathit{collect}(\widehat{\mathcal{RS}}, \widehat{ve}, \widehat{\sigma}, \widehat{binds}, \widehat{refs}) = \begin{cases} (\widehat{ve} \mid l_{\widehat{ve}},\ \widehat{\sigma} \mid l_{\widehat{\sigma}}) & \text{if } \begin{array}{l} \exists (v, \widehat{t}) \in \widehat{binds} :\ \widehat{ve}(v, \widehat{t}) \not\sqsubseteq \bot_{\widehat{\mathbf{D}}} \quad \vee \\ \exists \widehat{ref} \in \widehat{refs} :\ \widehat{\sigma}(\widehat{ref}) \not\sqsubseteq \bot_{\widehat{\mathbf{D}}} \\ \quad \text{where } \langle\langle \widehat{\varsigma}_i : \widehat{\varsigma}_i \in \widehat{\mathcal{RS}}\rangle, \varnothing, \varnothing, \varnothing\rangle \rightsquigarrow^* \langle\langle\rangle, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}\rangle \end{array} \\ (\widehat{ve}, \widehat{\sigma}) & \text{otherwise} \end{cases}$$

Figure 6.4: Garbage collection

extends a variable environment — that is, during the transition from an *apply* state to an *eval* state. Here, the analysis runs the risk of mixing new live values with old values that may have been unreachable until now. Therefore, the transition rule $\widehat{\text{ApplyClosGC}}$ shown in Figure 6.3 replaces the original rule $\widehat{\text{ApplyClos}}$ (see Figure 4.6 for the original rule).

$\widehat{\text{ApplyClosGC}}$ uses the *collect* function to run the garbage collector. This function restricts the variable environment $\widehat{ve}$ and store $\widehat{\sigma}$ of the*apply* state to the entries that are reachable from the root set. The root set consists of the *apply* state itself and the queue of states that the analysis has not visited yet (see Figure 4.7). This queue, however, is not part of the semantics because the queue is primarily needed in the implementation — the semantics as a conditional rewrite system can do without. To formally specify the garbage collector with the state queue nevertheless, I assume that all states of the queue at the time of a transition from an *apply* state reside in the set $\widehat{\mathcal{RS}}$.

In addition to the root set, the variable environment, and the store $\widehat{\text{ApplyClosGC}}$ also passes a set containing tuples of variables and points in time to *collect*. Each tuple consists of a variable bound by the lambda expression and the time of the state. That is, a tuple corresponds to an entry in the variable environment that must be updated when binding the arguments — these are the entries in the variable environment that may merge the new values with existing values.

Figure 6.4 shows *collect*. This function decides whether a garbage collection is necessary and either restricts the variable environment and store to the entries reachable through the root set $\widehat{\mathcal{RS}}$ or returns the unmodified variable environment and store. The operation $|$ carries out the restriction of functions (see Appendix A) and computes the restricted variable environment $l_{\widehat{ve}}$ and store $l_{\widehat{\sigma}}$.

Checking whether a garbage collection is necessary works as follows: The set $\widehat{binds}$ contains the entries in the variable environment $\widehat{ve}$ that will be updated by this *apply* state. If the variable environment $\widehat{ve}$ contains no information for these entries — $\widehat{ve}$ maps the entries to $\bot_{\widehat{\mathbf{D}}}$ — it is clear that no merging would occur. Hence, *collect* checks whether each of the given bindings from $\widehat{binds}$ maps to $\bot_{\widehat{\mathbf{D}}}$ and runs a garbage collection if this is not the case.

Finding the reachable entries precedes the restriction. The relation $\rightsquigarrow^*$ computes transitive reachability to determine the live variable environment entries

$$\rightsquigarrow: (\widehat{\mathbf{State}} + \widehat{\mathbf{D}})^* \times \mathcal{P}(\widehat{\mathbf{GCId}}) \times \mathcal{P}(\mathtt{Var} \times \widehat{\mathbf{Time}}) \times \mathcal{P}(\widehat{\mathbf{Ref}}) \rightarrow (\widehat{\mathbf{State}} + \widehat{\mathbf{D}})^* \times \mathcal{P}(\widehat{\mathbf{GCId}}) \times \mathcal{P}(\mathtt{Var} \times \widehat{\mathbf{Time}}) \times \mathcal{P}(\widehat{\mathbf{Ref}})$$

$$\frac{\omega \notin \widehat{\mathcal{S}}}{(\langle (lam, \widehat{\beta}, \widehat{t'})_\omega \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}'}, \widehat{\mathcal{S}'}, l'_{\widehat{ve}}, l_{\widehat{\sigma}})} \quad (\widehat{\text{GCClosU}})$$

$$\text{where} \begin{cases} \widehat{\mathcal{S}'} &= \widehat{\mathcal{S}} \cup \omega \\ l'_{\widehat{ve}} &= l_{\widehat{ve}} \sqcup \{(v, \widehat{t}) : \ v \in FV(lam) \ \wedge \ \widehat{t} = \widehat{\beta}(v)\} \\ \widehat{\mathcal{U}'} &= \widehat{\mathcal{U}} \S \big\langle \widehat{d}_i : \widehat{d}_i \in \bigsqcup_{v \in FV(lam)} \widehat{ve}(v, \widehat{\beta}(v)) \big\rangle \end{cases}$$

$$\frac{\omega \in \widehat{\mathcal{S}}}{(\langle (lam, \widehat{\beta}, \widehat{t})_\omega \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}})} \quad (\widehat{\text{GCClosS}})$$

$$\frac{\omega \notin \widehat{\mathcal{S}}}{(\langle (call, \widehat{\beta}, \widehat{ve'}, \widehat{\sigma'}, \widehat{t'})_\omega \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}'}, \widehat{\mathcal{S}'}, l'_{\widehat{ve}}, l_{\widehat{\sigma}})} \quad (\widehat{\text{GCEval}})$$

$$\text{where} \begin{cases} \widehat{\mathcal{S}'} &= \widehat{\mathcal{S}} \cup \omega \\ l'_{\widehat{ve}} &= l_{\widehat{ve}} \sqcup \{(v, \widehat{t}) : \ v \in FV(call) \ \wedge \ \widehat{t} = \widehat{\beta}(v)\} \\ \widehat{\mathcal{U}'} &= \widehat{\mathcal{U}} \S \big\langle \widehat{d}_i : \ \widehat{d}_i \in \bigsqcup_{v \in FV(call)} \widehat{ve}(v, \widehat{\beta}(v)) \big\rangle \end{cases}$$

$$\frac{\omega \notin \widehat{\mathcal{S}}}{(\langle (\widehat{proc}_\omega, \widehat{c}^*, \widehat{d}^*, \widehat{ve'}, \widehat{\sigma}, \widehat{t'})_{\omega'} \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}'}, \widehat{\mathcal{S}'}, l_{\widehat{ve}}, l_{\widehat{\sigma}})} \quad (\widehat{\text{GCApply}})$$

$$\text{where} \begin{cases} \widehat{\mathcal{S}'} &= \widehat{\mathcal{S}} \cup \omega' \\ \widehat{\mathcal{U}'} &= \widehat{\mathcal{U}} \S \widehat{proc}_\omega \S \big\langle \widehat{a}_i : \widehat{a}_i \in \bigsqcup \widehat{c}_i \big\rangle \S \big\langle \widehat{a}_i : \widehat{a}_i \in \bigsqcup \widehat{d}_i \big\rangle \end{cases}$$

$$\frac{\omega \in \widehat{\mathcal{S}}}{(\langle \widehat{\varsigma}_\omega \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}})} \qquad \frac{\widehat{d} \in \widehat{\mathbf{Bas}}_{Lang} \vee \widehat{d} = \mathtt{halt}}{(\langle \widehat{d} \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}})}$$

$$\frac{}{(\langle \widehat{ref} \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}} \S \langle \widehat{a}_i : \widehat{a}_i \in \widehat{\sigma}(\widehat{ref}) \rangle, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}} \sqcup \widehat{ref})} \quad (\widehat{\text{GCRef}})$$

$$\frac{\widehat{d}_\omega \in \widehat{\mathbf{Comp}}_{Lang} \qquad \omega \notin \widehat{\mathcal{S}}}{(\langle \widehat{d}_\omega \rangle \S \widehat{\mathcal{U}}, \widehat{\mathcal{S}}, l_{\widehat{ve}}, l_{\widehat{\sigma}}) \rightsquigarrow (\widehat{\mathcal{U}'}, \widehat{\mathcal{S}'}, l_{\widehat{ve}}, l'_{\widehat{\sigma}})} \quad (\widehat{\text{GCCompound}})$$

$$\text{where} \begin{cases} \widehat{\mathcal{S}'} &= \widehat{\mathcal{S}} \cup \omega \\ \widehat{refs} &= \textit{gc-follow}(\widehat{d}_\omega) \\ \widehat{\mathcal{U}'} &= \big\langle \widehat{d}_i : \ \widehat{d}_i \in \bigsqcup_{\widehat{r} \in \widehat{refs}} \widehat{\sigma}(\widehat{r}) \big\rangle \ \S \ \widehat{\mathcal{U}} \\ l'_{\widehat{\sigma}} &= \widehat{refs} \sqcup l_{\widehat{\sigma}} \end{cases}$$

Figure 6.5: Reachability relation

and live store entries. Figure 6.5 shows the definition of the reachability relation $\rightsquigarrow$. The relation simulates the collector used in the implementation. The relation relates tuples that correspond to the states of the garbage collector. A state consists of four components:

- The first component, the unvisited queue $\widehat{\mathcal{U}}$, is a vector that contains the unvisited states and unvisited abstract values.
- The second component, the seen ids $\widehat{\mathcal{S}}$, is a set of GC ids. This set contains all states and abstract values, that have already been visited during the garbage collection.
- The third component, the live entries $l_{\widehat{ve}}$, is a set of tuples. Each tuple consists of a variable and a point in time. These are the reachable variable environment entries.
- The fourth component, the live references $l_{\widehat{\sigma}}$, is a set of abstract references. These are the reachable store entries.

Before turning to the details of the reachability relation, I sketch the overall mode of operation of the relation: The garbage collection starts with an initial state that consists of empty sets for the live entries and references, an empty set of seen ids, and the states from the root set in form of a vector. In each step the reachability relation extracts the first element from the vector $\widehat{\mathcal{U}}$ and checks whether the seen id set $\widehat{\mathcal{S}}$ contains the GC id of this element. If the garbage collection already considered this value, the relation ignores this value and proceeds with the next value from the vector $\widehat{\mathcal{U}}$. The relation reaches its fixpoint when the vector for unvisited states and values is empty. For each type of new state or abstract value extracted from $\widehat{\mathcal{U}}$ the relation has a separate case. These cases add elements to $\widehat{\mathcal{U}}$, add GC ids to $\widehat{\mathcal{S}}$, and extend the sets of live environment entries and references. The process stops when $\widehat{\mathcal{U}}$ is an empty vector. That is, the relation describes a breadth-first search using $\widehat{\mathcal{U}}$ like a queue.

Consider the cases of the reachability relation shown in Figure 6.5. These cases correspond to Might's and Shivers's relation for touching values $\widehat{\mathcal{T}}$ [Might and Shivers, 2006]. This formulation, however, also takes the store into account and explicitly manages a queue of unvisited values — a preparation for the implementation.

The rules use a variable environment $\widehat{ve}$ and a store $\widehat{\sigma}$. This variable environment and store will be restricted in the final phase of the collection.

The case $\widehat{\text{GCClosU}}$ considers closures whose GC id is not in $\widehat{\mathcal{S}}$ and the case $\widehat{\text{GCClosS}}$ examines closures the collector has already visited. For closures that have already been visited the relation simply proceeds with the next element in the $\widehat{\mathcal{U}}$ vector. For unvisited closures the relation first computes the free variables of the lambda expression. The corresponding entries in the variable environment must be live. Therefore, $\widehat{\text{GCClosU}}$ adds these entries to $l_{\widehat{ve}}$. However, if these entries are reachable, the abstract values stored for this entry are also reachable and yield new reachable abstract values. Consequently, $\widehat{\text{GCClosU}}$ appends a vector with the abstract values found in the variable environment to $\widehat{\mathcal{U}}$.

The case $\widehat{\text{GCEval}}$ considers *eval* states and is similar to $\widehat{\text{GCClosU}}$. Again, free variables indicate live variable entries and the corresponding abstract values

in the variable environment are reachable and therefore require further inspection. *Apply* states do not contain expressions and therefore only add new values to $\widehat{\mathcal{U}}$. These values are the procedure values being applied and the argument vectors $\widehat{d}^*$ and $\widehat{c}^*$.

Right under the case for *apply* states, Figure 6.5 shows two cases without names. The left case concerns states that have been visited before. The case on the right-hand side is responsible for basic values and the `halt` continuation: These values are uninteresting for the garbage collector and the rule just skips these values in $\widehat{\mathcal{U}}$. This case, however, is necessary because the cases that extend $\widehat{\mathcal{U}}$ (such as $\widehat{\text{GCApply}}$) simply add all values to $\widehat{\mathcal{U}}$ without checking whether the type of the values is relevant for the collector. Filtering the abstract values before adding them to $\widehat{\mathcal{U}}$ would make the cases $\widehat{\text{GCClosU}}$, $\widehat{\text{GCEval}}$, and $\widehat{\text{GCApply}}$ more complex to read and understand.

If the first element in $\widehat{\mathcal{U}}$ is an abstract reference case $\widehat{\text{GCRef}}$ applies. This reference is assumed to be live and the rule therefore adds this reference to $l_{\widehat{\sigma}}$. The abstract values in the store identified by the reference are subject to further inspection by the reachability relation and therefore become part of $\widehat{\mathcal{U}}$.

$\widehat{\text{GCCompound}}$ is the most interesting case. This case considers compound values and exploits the representation of compound values (see Sections 4.4.1 and 4.4.2). Recall that compound values hold information specific to the value type (e. g. the vector length or a record type) and an abstract reference for each field. That is, the garbage collector can extract the necessary information from the compound value:

- The collector adds all references that belong to the compound value to the set of live references.
- With the references at hand the collector traces the abstract values reachable through the compound value and adds them to $\widehat{\mathcal{U}}$.
- A compound value is equipped with a GC id. That is, if the collector discovers this compound value again it can cut off the search without examining the fields.

The case $\widehat{\text{GCCompound}}$ does exactly this. The function *gc-follow* is the counterpart of `gc-follow` (see Section 6.2.4) and returns the set of abstract references a compound value uses to store the field values.

6.4.2 Abstract values with GC marks

The reachability relation $\rightsquigarrow$ and the collection function *collect* define the garbage collector in a form appropriate for an efficient implementation. This section describes how to turn the specification into code.

The reachability relation memorizes the GC ids of objects already encountered using the set $\widehat{\mathcal{S}}$. All cases of $\rightsquigarrow$ check whether the GC id of the object in question is a member of $\widehat{\mathcal{S}}$. In the implementation $\widehat{\mathcal{S}}$ exists in this form: Each state or abstract value contains an additional component, the so-called *GC mark*, that distinguishes seen from unseen objects. A GC mark is a record with just one field that stores the *GC time*.

The GC time advances only when the analysis runs a garbage collection. Figure 6.6 shows the code for GC marks. To check whether an object was

```
(define-record-type gc-mark :gc-mark
  (really-make-gc-mark time)
  gc-mark?
  (time gc-mark-time set-gc-mark-time!))

(define (make-gc-mark)
  ;; The initial value -1 ensures that this mark
  ;; is unseen on the first GC.
  (really-make-gc-mark -1))

(define *gc-time* 0)

(define (tick-gc-time!)
  (set! *gc-time* (+ 1 *gc-time*)))

(define (seen-gc-mark? gc-mark)
  (= (gc-mark-time gc-mark) *gc-time*))

(define (seen? thing)
  (cond
   ((state? thing)
    (seen-gc-mark? (state-gc-mark thing)))
   ((abs-closure? thing)
    (seen-gc-mark? (abs-closure-gc-mark thing)))
   ((abs-compound? thing)
    (seen-gc-mark? (abs-compound-gc-mark thing)))
   (else #t)))

(define (have-seen-gc-mark! gc-mark)
  (set-gc-mark-time! gc-mark *gc-time*))
```

Figure 6.6: Implementation of GC marks

visited before the collector calls the procedure `seen?`, which extracts the GC mark of a state, closure, or compound value and compares the GC time in this record with the current GC time stored in the global variable `*gc-time*`. If both are equal, the collector already visited this object during this collection. To register an object as seen, the collector calls `have-seen-gc-mark!` to update a GC mark to hold the current GC time.

I borrowed this idea from the implementation of garbage collectors for programming languages: Distinguishing live objects from unreachable objects using a marking bit is a popular technique and was originally introduced by the first Lisp implementations [*Jones and Lins*, 1996]. In the context of a flow analysis the marking bit becomes an integer, which eliminates the need for reseting the marking bit. This approach is simpler and faster than traversing the complete abstract data just to reset one bit.

Note that the garbage collector for the flow analysis continues with a restricted variable environment and a restricted store and leaves the old variable environment and store untouched.

The representation of the remaining components of a state tuple is straightforward. The vector $\widehat{\mathcal{U}}$ becomes a regular queue. For the sets of live variable

environment entries $l_{\widehat{ve}}$ and live references $l_{\widehat{\sigma}}$ hash tables are a suitable representation because testing for the membership of an element is fast. This operation is important during the last phase of the collection because it restricts the variable environment and the store.

The garbage collector often computes the free variables of an expression. Garbage collection may occur frequently with the result that the garbage collector spends a lot of time traversing the syntax tree and computing sets of free variables. This, however, is easy to fix: The implementation manages an id table (see Section 6.2.5) that maps syntax nodes to sets of free variables. The procedure `free-variables` computes the free variables using the id table as a cache. That is, for each syntax node the collector computes the set of free variables exactly once.

Figure 6.7 shows an excerpt from the implementation of the global garbage collector.[3] The procedure `garbage-collect` starts the collection with the given root set, the variable environment, and the store. The global variable `*live-binds*` is the id table that holds the live variable environment entries (these entries are called *abstract bindings* in the implementation). `Garbage-collect` adds all states in the root set to the queue of unvisited values `unvisited-values`. The procedure `trace!` implements the case analysis of the reachability relation. Figure 6.7 only shows the cases for closures and compound values. The code corresponds directly to the formal specification of Figure 6.5: The set $\widehat{\mathcal{S}}$ exists in form of the GC marks that `trace!` checks and updates. `Register-live-bind!` is the correspondent of adding a live entry to $l_{\widehat{ve}}$. The operations on $\widehat{\mathcal{U}}$ directly turn into the queue operations `enqueue!` and `dequeue!`.

6.4.3 Garbage Collection and variable environment

So far a discussion on the representation of the variable environment has been postponed because of the complex interplay of variable environments and garbage collection. After having discussed the garbage collector it is now time to consider an implementation strategy for the variable environment. The representation is geared towards the global garbage collector.

Shivers shows that variable environments grow monotonically [*Shivers*, 1991]: Transitions from *apply* states always join an existing variable environment with new entries. Shivers exploits this observation by using a timestamp mechanism and a single global variable environment. The timestamp (an integer) replaces the component for the variable environment in each state. The entries of a variable environment reside in a list that is globally accessible. Extending a variable environment implies the following operations: First, the implementation updates the global table to hold the new entry or the new abstract value. Second, the time on the timestamp clock advances. Checking whether two states were created with the same variable environment now is trivial since it is only necessary to compare the timestamps: An earlier timestamp indicates a younger variable environment and thus also is smaller in the sense of $\sqsubseteq$.

The advantages of the timestamp approach in comparison with a naive implementation that associates a table or association list with each state are as follows:

- Saves spaces. There is just one global table for all states.

[3] For a better readability I elide all code that is related to the registration of store references.

```
(define (trace! unvisited-queue venv store val)
  (cond
   [...]
   ((abs-closure? val)
    (if (not (seen-gc-mark? (abs-closure-gc-mark val)))
        (let* ((node (abs-closure-node val))
               (free-vars (free-variables node)))
          (have-seen-gc-mark! (abs-closure-gc-mark val))
          (for-each
           (lambda (var)
             (let* ((bind-time (lookup-binding-env
                                (abs-closure-benv val) var))
                    (bind (make-abs-bind var bind-time))
                    (val (lookup-variable-env venv bind)))
               (register-live-bind! bind)
               (for-each (lambda (sv)
                           (enqueue! unvisited-queue sv))
                         val)))
           free-vars))))

   ((abs-compound? val)
    (let ((follow-proc (abs-compound-gc-follow-proc val))
          (gc-mark (abs-compound-gc-mark val)))
      (if (not (seen-gc-mark? gc-mark))
          (let ((reachable (follow-proc val)))
            (have-seen-gc-mark! gc-mark)
            (for-each (lambda (v)
                        (enqueue! unvisited-queue v))
                      reachable)))))
   [...]))

(define (garbage-collect root-set venv store)
  (tick-gc-time!)
  (set! *live-binds* (make-abs-bind-set))
  (let ((unvisited-values (make-queue)))
    (for-each (lambda (v)
                (enqueue! unvisited-values v))
              root-set)
    (let lp ()
      (if (queue-empty? unvisited-values)
          *live-binds*
          (let ((abs-val (dequeue! unvisited-values)))
            (trace! unvisited-values venv store abs-val)
            (lp))))))
```

Figure 6.7: Excerpt from the garbage collector

- Comparison is fast. Comparing two variable environments using $\sqsubseteq$ occurs frequently during the termination check.

The timestamp approach, however, has one small drawback that concerns the interpretation of the information stored in the visited set: With the timestamp implementation described above it is no longer possible to reconstruct the variable environment as it was at the time a certain state transition occurred. During the inspection of a certain state it may be possible to observe entries or abstract values in the variable environment that were not present at the time the flow analysis carried out the state transition. This may be confusing for the user, especially when stepping through the transitions retroactively or during the process of debugging. However, the abstract values bound to the (relevant) variables that are evaluated during an *eval* state may be reconstructed by looking at the successor state: The successor is an *apply* state that contains the result of the variable evaluation in its argument list.

For the analysis of realistic programs the timestamp approach is ideal. Using the timestamp approach in the presence of a garbage collector, however, yields the following problem: The variable environment only grows monotonically until a collection occurs. The collection removes unreachable entries from the variable environment and consequently the restricted variable environment is not larger in the sense of $\sqsubseteq$. Subsequent updates of the variable environment then again add values: These values are either known, that is a non-collecting environment would already contain this value or these are new values. It is important to distinguish both types of updates to ensure termination. Thus, the variable environment keeps multiple sets of abstract values for each entry: A set always contains all values since the last garbage collection.

The variable environment consists of two parts: An eq hashtable that stores all entries and record type that wraps an integer number — the timestamp. The clock for timestamps advances only when the analysis merges new values into the environment. If the analysis updates an environment with a value for which the corresponding entry already contains an approximation the time does not advance. That is, the timestamp counts the number of updates that add new information to the environment.

Each entry in table corresponds to an entry in the variable environment and contains an association list that maps a *GC time* to the abstract value. The GC time is a counter that tracks the number of garbage collections performed. Assume that no garbage collector occurred so far, then an entry in the variable environment is a list with one element:

```
'((0 #{Apsval integer 23} #{Apsval integer 42}))
```

This entry maps the GC time zero to the abstract PreScheme values for the (exact) integer numbers 23 and 42. If the analysis updates the environment with $\top_{Int}$ this adds information to the environment and the timestamp clock advances. The resulting entry contains the new value for GC time zero:

```
'((0 #{Apsval integer 23} #{Apsval integer 42} #{Apsval integer #f}))
```

The string `#{Apsval integer #f}` is the external representation of $\top_{Int}$. If the garbage collector is turned off or no collection is necessary only the entry for GC time zero grows. Thus, the variable environment corresponds to the original timestamp approach.

Assume that the analysis performs a garbage collection and the entry of the example above becomes unreachable. First, the GC time advances from zero to one. Also assume that the entry depicted above becomes unreachable. The variable environment represents this situation by adding a new entry that maps the value to ⊥ (represented as the empty list) for GC time one:

```
'((1) (0 #{Apsval integer 23} #{Apsval integer 42} #{Apsval integer #f}))
```

Thus, if the analysis evaluates the variable that corresponds to this entry under the current GC time the variable environment returns ⊥. Subsequent updates of this variable merge the new value with the ⊥ value from the latest GC time.

6.4.4 Splitting the root set

An analysis of a large program often leads to a situation as follows: The visited set contains many entries (a few thousand states), garbage collection occurs frequently (almost on every second state transition), and the queue of unvisited states is long (a few hundred states). Computing the reachability relation is expensive because the heap (variable environment and store considered together) contains many abstract values and the root set is also large. Recall that the states in the unvisited queue belong to the root set. This means that from collection to collection the root set changes only slightly: Many states from the state queue still reside in the queue and belong to the root set. The garbage collector, however, computes the reachable relation for all states and therefore traces a large portion of the root set again and again.

A simple idea to avoid this situation goes back to splitting the root set into a number of sets — each set contains exactly one state from the original root set. Then the collector computes each reachability relation separately, and stores the result along with the state in the state queue. This approach, however, is difficult to implement: Instead of a single set of live entries the result now consists of a sequence of such sets. That is, to restrict the variable environment it is necessary to search all these sets for an entry. Plus, these sets require a considerable amount of memory.

I have implemented the split root sets just described. The experiments I conducted with this technique lead to the following results: The improvement in performance accounts for five percent in average. Here, investigating more sophisticated implementation techniques is necessary.

6.5 Experimental results

The tables in Figures 6.8 and 6.9 show the analysis times for some Scheme programs of the Gambit benchmark suite [*Feeley*, 1998]. Most of the benchmark programs are Scheme versions of the classical Lisp benchmark programs introduced by Richard Gabriel [*Gabriel*, 1985]. The columns of the table show the following information:

size The size of the program given as the number of user-defined procedures and the number of continuation lambda expressions.

GC The garbage collection policy and the number of collections performed during the analysis. The interval policy performs a garbage collection on every tenth transition from an *apply* state.

name	size*	GC		states	checks	queue	time
		never	—	126	337	6	0.33
`ack`	2/15	demand	6	71	84	3	0.20
		interval	9	86	141	10	0.20
		never	—	146	337	6	0.36
`ctak`	8/29	demand	5	118	200	2	0.30
		interval	11	118	200	2	0.33
		never	—	185	700	5	0.66
`cpstak`	7/17	demand	11	146	408	5	0.70
		interval	13	139	362	5	0.63
		never	—	89	250	5	0.20
`fib`	2/11	demand	9	85	217	5	0.30
		interval	8	89	242	5	0.23
		never	—	350	1,619	11	0.11
`fibc`	8/35	demand	9	154	314	3	0.50
		interval	27	292	1,048	10	0.11
		never	—	536	2,375	12	1.40
`primes`	6/49	demand	59	550	2,690	15	2.62
		interval	47	538	2,392	11	2.15
		never	—	173	647	5	0.50
`tak`	2/18	demand	13	168	601	5	0.60
		interval	16	175	664	5	0.60
		never	—	397	1,750	8	1.01
`takl`	3/35	demand	16	348	1,302	5	1.20
		interval	32	352	1,361	5	1.25
		never	—	224	355	4	0.53
`array`	8/40	demand	9	224	355	4	0.76
		interval	23	270	540	8	0.90
		never	—	1,399	8,316	18	0.68
`destruc`	12/111	demand	59	862	2,783	22	0.65
		interval	127	1,501	9,469	10	1.56
		never	—	583	1,055	8	0.23
`fft`	8/108	demand	27	714	1,763	8	0.40
		interval	53	605	1,225	8	0.35
		never	—	2,851	33,707	33	1.80
`deriv`	11/125	demand	93	1,087	4,656	31	0.79
		interval	67	786	2,335	28	0.43
		never	—	62	39	3	0.10
`divrec`	2/11	demand	6	114	39	3	0.13
		interval	6	100	39	3	0.12
		never	—	947	3,941	12	0.34
`triangl`	6/93	demand	24	602	1,457	7	0.26
		interval	53	592	1,460	10	0.25
		never	—	10,619	190.6	46	20.03
`trav`	39/295	demand	308	5,871	52.3	49	8.52
		interval	597	6,840	78.8	44	13.15

* The number of user-defined procedures and continuation lambda expressions.

Figure 6.8: Analysis times for Scheme programs 1

name	size	GC		states*	checks†	queue	time
pi	22/265	never	—	8,966	180.2	47	9.42
		demand	140	2,164	13.3	23	1.87
		interval	222	2,526	17.6	29	2.27
pnpoly	3/164	never	—	3,358	35.6	29	1.72
		demand	62	1,056	4.8	16	0.62
		interval	53	599	1.6	20	0.34
paraffins	30/273	never	—	7,626	103.8	56	8.61
		demand	525	5,147	47.7	61	12.23
		interval	342	4,058	34.5	89	7.91
mbrot	8/58	never	—	375	0.9	6	0.12
		demand	19	301	0.5	6	0.14
		interval	48	533	2.0	10	0.23
nqueens	10/80	never	—	1,920	21.6	23	1.16
		demand	158	1,379	10.4	23	1.26
		interval	120	1,385	10.5	10	1.70
ray	37/359	never	—	6,573	103.6	58	6.47
		demand	99	2,004	8.0	14	1.77
		interval	112	1,246	4.1	10	0.73
perm9	11/62	never	—	193	0.2	5	0.40
		demand	9	184	0.2	5	0.60
		interval	49	540	1.6	10	0.26
maze	79/825	never	—	62,241	3,045.4	184	298.88
		demand	503	7,933	62.4	30	11.79
		interval	991	11,205	159.2	10	25.62
lattice	46/305	never	—	10,494	205.8	86	23.80
		demand	1,224	9,122	175.7	144	40.54
		interval	417	4,874	46.7	79	8.13
puzzle	22/319	never	—	4,623	44.0	30	3.48
		demand	136	2,519	12.7	15	1.92
		interval	289	3,276	16.4	10	2.55
boyer	26/342	never	—	12,734	217.7	81	21.90
		demand	1,426	13,105	246.6	188	44.51
		interval	724	8,677	106.6	135	16.60
simplex	32/647	never	—	26,161	682.6	118	60.90
		demand	506	5,554	41.2	47	18.89
		interval	960	11,025	146.4	121	59.34
matrix	99/734	never	—	56,690	2,284.1	229	239.80
		demand	6,204	43,005	1,802.2	329	545.53
		interval	3,114	36,333	1,194.6	280	251.20
graphs	60/408	never	—	13,485	2,229.6	110	19.66
		demand	1,336	10,985	1,508.6	202	67.70
		interval	1,068	12,267	2,024.8	178	51.88
earley	97/828	never	—	51,575	1,855.7	213	378.13
		demand	3,606	28,895	715.4	383	581.54
		interval	1,953	22,957	434.4	279	249.82

* The number of user-defined procedures and continuation lambda expressions.
† The number of checks divided by 10^3.

Figure 6.9: Analysis times for Scheme programs 2

name	size	states	checks	queue	time [min]
`Twospace GC`	37/2202	125,307	6,534	467	6.57
`Scheme 48 VM`	185/11,082	785,668	42,354	2,257	218.54

Figure 6.10: PreScheme benchmarks

states The number of visited states.

checks The number of state comparisons using $\sqsubseteq$. Note the numbers in table 6.9 are divided by 10^3.

queue The maximum number of states waiting in the queue.

time The total analysis time in seconds. The times for the small benchmark programs of table 6.8 are the average of three runs.

The analysis ran on Scheme 48, version 1.8, including the patch for eq hashtables (see Section 6.2.5). Scheme 48 ran with a total of 2 GB heap space using a single core on a MacPro with two Intel Dual-Core Xeon processors running at 2.66 GHz. The analysis used the following configuration:

Collector Global garbage collector with marking.

Time A 0CFA time abstraction.

Variable environment Global variable environment using timestamps (as described in Section 6.4.3).

Binding environment Based on sorted association lists (as described in Section 6.2.2).

Visited set Based on eq hashtables (as described in Section 6.2.5).

State queue State queue using eq hashtables.

Relation cache Configured to cache the last 200 comparisons of binding environments.

Figure 6.10 shows two benchmark results for substantial PreScheme programs: The Twospace Garbage Collector of Scheme 48 and the complete virtual machine itself. The size of these programs is measured after the transformational compiler inlined the single-use procedures. Therefore the number of user-defined procedures may appear low. Note that the time in this table is given in minutes.

The benchmark results suggest the following conclusions: The cost for garbage collection is significant, especially for larger analyses. For some programs the cost of garbage collection outweighs the time saved by a smaller state space. The benchmarks show that the coherence between the effect garbage collection and analysis time is complex: Some benchmarks, such as `maze`, `boyer`, and `paraffins`, profit from garbage collection above average. Here, the collector reduces the state space considerably. For other programs (i. e. `nqueens` and `graphs`) the state space is almost as big as in an analysis without collection.

The criterion that decides when to perform a garbage collection has a large influence on the overall time. Consider the results for the “on demand” strategy:

Especially for larger analyses such as `matrix`, `earley`, and `lattice`, the number of collections is high: Often the number of collections doubles in comparison with the interval strategy which runs the collector on every tenth transition. That is, collections occur at a frequency higher than every tenth state. In fact, many of the benchmark programs run a collection on every transition from an *apply* state. This occurs often with programs that use lists. The analyses for such programs usually contains many abstract references to pairs and traversing a list yields to many merges in the variable environment, which consequently triggers a garbage collection. In summary, running the collector in an interval of ten state transitions seems to be a middle course that improves the speed of the analysis for almost all test cases. Furthermore, the analysis does not run risk of spending too much time on garbage collection.

Consider the number of termination checks carried out for the benchmarks. The number of checks for the Scheme benchmark are much higher than the checks carried out during the analysis of PreScheme programs — even though the PreScheme programs are larger and more complex then the Scheme programs. This effect is due to the size of hashtable used in the implementation of the visited set: The PreScheme benchmarks use a very large hashtable with 2^{17} entries while the hashtable for the Scheme benchmarks only has 2^{15} entries.

Note that analyses that have a many waiting states in unvisited queue such as `matrix` and `earley` only benefit below average from the garbage collector. Many states in the queue means that the root set is large and therefore leads to longer collection times. Furthermore, the set of reachable values is larger if many states wait in the queue. These properties make garbage collection less effective.

In summary, the benchmarks show that the analysis of realistic and complex programs benefits from garbage collection. The benefits are satisfying even though the effort necessary to implement an efficient collector is high.

6.6 Testing and Debugging

Testing and debugging a flow analysis is troublesome. The experimental results show that even for small programs the state space grows very quickly. An additional problem arises from the fact that tracking the relation between cause and effect is especially complicated: The cause of a problem, such as a wrong abstract value returned by an erroneous abstraction function may be introduced early in the analysis but may surface and cause a problems in a very late phase of the analysis. Here, systematic debugging techniques [*Zeller*, 2005] are indispensable. I used the following tools to test and debug the flow analysis implementation:

Log and transition graph In debug mode, the implementation writes an extensive log file. This log includes all relevant information such as the transition rules applied, the results of argument evaluation, regular snapshots of the variable environment and store, a trace of the termination check, and the code under consideration. An analysis of Twospace-GC (see previous section), for example, has about 125,000 states and produces a 700 mega-byte log file. It is important to navigate the log easily and search the log for certain values,

program labels, or states. Therefore, the analysis assigns an unique id to each state, closure, binding environment, and variable environment.

Also, the analysis generates a graph that depicts the state transitions every fifty states: The graph is given as GML file which may be viewed using yEd [*Wiese et al.*, 2001]. Given a failed state the graph facilitates searching backwards through the transitions and helps finding the source of the problem.

Finding similar states Errors that cause the analysis to loop infinitely are hard to find and often go back to small mistakes in the implementation of $\sqsubseteq$. To identify the problem I developed a tool called *state-diff* that searches the visited set for states that have a high chance of being related by $\sqsubseteq$. The tool lists all relevant differences that cause the approximation check to fail. The additional information printed by state-diff, however, almost doubles the size of the log file.

Assertions Almost every abstraction function and every function that operates on the variable environment, the store, or the visited set uses assertions heavily to recognize a problem as soon as possible. The implementation includes its own Scheme macro for assertions that allows to check pre-conditions, post-conditions, and includes many options to report failures. Using assertions proved to be very helpful for the development.

Test suite The analysis contains a test suite that covers the basic functionality such as store operations, argument evaluation, and variable environment operations. Tests that involve a complete program and analysis are complicated to set up and check: It is almost impossible to check all aspects of the information computed by the analysis. Here, only spot tests are feasible. Such tests check the following: Does the visited set contain states for each point in the program that is supposed to be visited? Do the argument vectors of the *apply* states contain the expected values? Do the primops return the correct abstract values?

These techniques proved helpful to identify a lot of problems during development. Still, debugging and testing the flow analysis was among the most time-consuming and work-intensive parts of this dissertation.

Chapter 7

Conclusion

This dissertation shows how to scale control-flow analyses for higher-order languages up to complete, fully-fledged programming languages and compute the flow analysis of realistic programs.

7.1 Review

The flow-analysis framework operates on the intermediate language of Scheme 48's transformational compiler. In a first preparatory step, Chapter 2 defines an operational small-step semantics for the intermediate language. The semantics focuses on procedure application and omits the definition of denotable values. This semantics is the basis for the systematic design of computable abstract semantics that specify the flow analysis. The abstract semantics consists of three parts: core semantics and semantics for PreScheme and Scheme. The semantics for the core of the intermediate language merely deals with procedure application, the global variable environment, and the basic primitive operations that implement local recursion and conditional evaluation.

The abstract semantics for PreScheme and Scheme add the definitions for denotable values and primitive operations that are necessary to cover the PreScheme and Scheme language. The specifications include support for procedure calls with variable arity, multiple return values, precise abstraction for heap objects such as records, pairs, and vectors, abstractions for all simple value types, and most primitive operations of the Scheme 48 system and the PreScheme compiler. These coherent and complete specifications of the flow-analyses for PreScheme and Scheme are a core contribution of this dissertation. The analyses compute the control flow and detailed information on the values used in the program. Thus, the analysis results may be used to drive several optimizations.

A correctness relation defines the semantic correctness of the flow analysis by relating entities of the concrete semantics with abstract entities that are a valid abstraction. To test, trace, and inspect the specification of the flow analysis I use an executable model of the semantics. The model implements the concrete semantics, the flow-analysis semantics, and the correctness relation as a term rewrite system. For a given program the model computes all reduction steps of both semantics and relates the results using the correctness relation. Thus, testing, tracing, and visualizing and the mode of operation of the analysis

becomes possible. This model is a core contribution of this dissertation and an important preparatory step for the implementation of the analysis in the real compiler.

The implementation developed as part of this dissertation operates as part of the transformational compiler, supports all features of PreScheme and Scheme, and is capable of analyzing realistic programs. Performance is the main challenge for such an implementation. This dissertation contributes a systematic investigation and design of the representations for the semantic domains and a global garbage-collection technique for flow analyses. Both, sophisticated representations and the garbage collector improve the performance of the analysis. The novel global garbage collector always considers the complete state of the analysis and enables an efficient implementation of the variable environment using timestamps.

7.2 Future work

The flow analysis implementation in Scheme 48 opens a wide range of opportunities for investigations on applications of flow-analysis results under realistic conditions:

Flow language This dissertation describes the flow language, the language for describing flow information syntactically, only briefly and in an ad-hoc manner. This idea, however, needs further investigation and a theoretical foundation. Currently, users can specify the flow behavior of a procedure (or a whole library) in terms of the flow language. More interesting, however, is the following question: Is it possible to derive the a flow language description from flow information actually computed by the analysis? Certainly, this process would require abstracting the computed flow information to achieve a version that carries less information and results in a more concise (but still correct) description. There are two uses for generating a readable description automatically: First, the analysis could suggest this description to the user for an interactive refinement. Second, the analysis could generate simple descriptions for libraries. Such a description could be stored in a file and reused for each program that uses this library. The result would be a system similar to Flanagan's componential analysis with constraint compression [*Flanagan and Felleisen*, 1999].

For abstracting flow information, Dubé's techniques for a demand-driven analysis may be helpful [*Dubé*, 2002]. Additionally, developing an abstraction based on the systematic construction of Galois connections [*Nielson et al.*, 2005] should be considered.

Executable model The executable model of the semantics as presented in Section 5 does not model the complete flow-analysis semantics of PreScheme as it does not include all primops and some value types. A complete model, however, is desirable as it would allow tracing realistic test cases or even serve as debugging aid.

Currently the executable model checks the correctness relation and simply returns a boolean. However, if the check returns false information on the circumstances that lead to the negative result are desirable. This would be especially helpful to test new abstraction functions.

Generating graphs that show the traces of both semantics and the connection via the correctness relation currently involves manual work. To ease this process it would be helpful to connect the model to a graph-drawing software that supports layered graphs.

Applications Flow analysis is not an end in itself. The information acquired by the analysis are useful to program optimization. Currently, the implementation only uses flow information for inline expansion [*Jagannathan and Wright*, 1996; *Ashley*, 1997] and the sophisticated super-β inlining [*Might and Shivers*, 2007]. My implementation of inline expansion, however, is not complete: It computes a large set of chances to inline and implements the actual inline expansion but lacks an appropriate heuristics for deciding which inlining chances to take. I conjecture that using flow information for this decision may be helpful.

For the native-code compiler an analysis that allows to eliminate the type checks of primitive operations seems promising. The work of Jagannathan and Wright [*Jagannathan and Wright*, 1995] shows that, on basis of flow information, the number of necessary type checks can be reduced significantly. In this context, revisiting Shivers's type recovery algorithm [*Shivers*, 1991] is also interesting. Also, flow analysis can discover opportunities for an unboxing optimization.

Implementation The implementation of the analysis for Scheme 48 could be improved along the following lines:

Storing the analysis results in a file is desirable for pragmatic reasons. For large analyses this would save time when developing applications that use these results.

Debugging the analysis involves reading very large log files that are hard to navigate. Here, a graphical debugging tool that makes it easy to follow state transitions, inspect and compare environments, or even check given correctness conditions would be a relief.

Multi-core processor system started becoming a standard in recent years. The current implementation does not benefit from this development as the underlying Scheme implementation does not support multiprocessor system. A recent development version of Scheme 48, however, uses multiple threads at operating-system level to evaluate Scheme code and therefore benefits of multi-core processors [*Frese*, 2006]. I conjecture that the analysis could benefit from a parallel evaluation. The worklist algorithm at the core of the analysis is straightforward to parallelize and the implementation uses very few mutable shared data structure that must be synchronized.

The analysis of the source code for Scheme 48 virtual machine (about 20,000 lines of code) consists of 750,000 states. This amount of states is close to the limits of the current implementation. The heap space required during the analysis amounts to 2 GB and 4 GB in total for both heap spaces of Scheme 48's two-space collector. Using, the Scheme 48 on a 64-bit system, however, is no relief: Each Scheme value doubles in size and as a result even more memory is necessary. Thus, a better implementation of id tables that works with the new generational garbage collector of Scheme 48 is necessary. This would enable to apply the analysis to programs that are larger and more complex than the virtual machine.

Appendix A

Notation

The notation used in this work generally follows the notation used in [*van Leeuwen*, 1990] and [*Nielson et al.*, 2005].

Vectors Vectors are ordered sequences of a fixed length that may be accessed with a 1-based index. $\langle a_1, \ldots, a_n \rangle$ denotes a vector of length n. The operator § concatenates the elements of a vector:

$$\langle a_1, \ldots, a_n \rangle \S \langle b_1, \ldots, b_m \rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$$

The elements of A^* are vectors (of unspecified length) of elements in A. A lowercase letter with a star is a member of A^*, i. e. $a^* \in A^*$. The function *map* applies a function to each member of a vector and returns a new vector with the results. $v \downarrow i$ accesses the element at index i in vector v.

$\langle e_i : P(e_i) \rangle$ creates a vector with elements e_i; for each element e_i for which the predicate $P(e_i)$ holds. The order of the elements is unspecified as well as the length of the vector.

Functions The syntax $[a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots]$ denotes the construction of a function that maps a_i to b_i. $[]$ denotes the empty function for which $\mathrm{dom}([]) = \varnothing$ holds. $f[a \mapsto b]$ constructs a new function by extending f:

$$f[a \mapsto b](x) = \begin{cases} b & x = a \\ f(x) & \text{otherwise} \end{cases}$$

The $|$ operator restricts the domain of a function to parameters given in a set:

$$f|R \Leftrightarrow [x_i \mapsto f(x_i)] \quad \forall x_i \in \mathrm{dom}(f) \cap R$$

Appendix B

Semantic Correctness

This chapter contains the proof of the semantic correctness of the flow analysis as defined by the correctness relation $\mathcal{R}$ (see Section 4.3).

In preparation for the proof, four lemmas are necessary. The first lemma states that merging abstract values does not affect the correctness:

Lemma 2. *If $d \ \mathcal{R} \ \widehat{d}$, $d \in \mathbf{D}$, $\widehat{d}, \widehat{d}' \in \widehat{\mathbf{D}}$ then $d \ \mathcal{R} \ (\widehat{d} \sqcup \widehat{d}')$.*

Proof. Since $d \ \mathcal{R} \ \widehat{d}$ there exists an abstract value $\widehat{v} \in \widehat{d}$ such that $d \ \mathcal{R} \ \widehat{v}$ and this value also exists in $\widehat{d} \sqcup \widehat{d}'$. □

The following lemma states that updating the concrete and abstract variable environments with a variable and point in time preserves the semantic correctness:

Lemma 3 (Updating variable environments)**.** *Let $ve \in \mathbf{VEnv}$, $\widehat{ve} \in \widehat{\mathbf{VEnv}}$, $t \in \mathbf{Time}$, $\widehat{t} \in \widehat{\mathbf{Time}}$, $d \in \mathbf{D}$, and $\widehat{d} \in \widehat{\mathbf{D}}$ such that $t \ \mathcal{R} \ \widehat{t}$ and $d \ \mathcal{R} \ \widehat{d}$. If $ve \ \mathcal{R} \ \widehat{ve}$ and*

$$\begin{aligned} ve' &= ve[(v,t) \mapsto d] \\ \widehat{ve}' &= \widehat{ve} \sqcup [(v,\widehat{t}) \mapsto \widehat{d}] \end{aligned}$$

then $ve' \ \mathcal{R} \ \widehat{ve}'$.

Proof. From $\widehat{ve} \sqcup [(v,\widehat{t}) \mapsto \widehat{d}]$ follows that $\widehat{ve}'(v,\widehat{t}) = \widehat{ve}(v,\widehat{t}) \sqcup \widehat{d}$ which by Lemma 2 means that $ve(v,t) \ \mathcal{R} \ \widehat{ve}'(v,\widehat{t})$. By $ve \ \mathcal{R} \ \widehat{ve}$ it is clear that

$$\forall (v,t) \in \mathrm{dom}(ve) : \forall \widehat{t} \in \widehat{\mathbf{Time}} : t \ \mathcal{R} \ \widehat{t} \Rightarrow ve(v,t) \ \mathcal{R} \ \widehat{ve}(v,\widehat{t})$$

Both facts considered together give $ve' \ \mathcal{R} \ \widehat{ve}'$. □

The transition rules for *apply* states also extend the binding environments. When extending the binding environments for a variable, the resulting binding environments are related by $\mathcal{R}$:

Lemma 4 (Updating binding environments)**.** *Let $\beta \in \mathbf{BEnv}$, $\widehat{\beta} \in \widehat{\mathbf{BEnv}}$, $t \in \mathbf{Time}$, and $\widehat{t} \in \widehat{\mathbf{Time}}$ such that $t \ \mathcal{R} \ \widehat{t}$. If $\beta \ \mathcal{R} \ \widehat{\beta}$ and*

$$\begin{aligned} \beta' &= \beta[v \mapsto t] \\ \widehat{\beta}' &= \widehat{\beta}[v \mapsto \widehat{t}] \end{aligned}$$

it follows that $\beta' \ \mathcal{R} \ \widehat{\beta}'$.

Proof. By $\beta \mathcal{R} \widehat{\beta}$ it is clear that $\forall u \in \mathrm{dom}(\beta) : \beta(u) \mathcal{R} \widehat{\beta}(u)$. By $t \mathcal{R} \widehat{t}$ it follows that $\beta'(v) \mathcal{R} \widehat{\beta}'(v)$. Considered together, this means $\beta' \mathcal{R} \widehat{\beta}'$. □

Updating the concrete and abstract stores yields new concrete and abstract stores which are connected by the semantic correctness relation $\mathcal{R}$:

Lemma 5 (Updating the store)**.** *Let* $ref \in \mathbf{Ref}$*,* $\widehat{ref} \in \widehat{\mathbf{Ref}}$*,* $d \in \mathbf{D}$*, and* $\widehat{d} \in \widehat{\mathbf{D}}$ *such that* $ref \mathcal{R} \widehat{ref}$ *and* $d \mathcal{R} \widehat{d}$*. If* $\sigma \mathcal{R}_{Store} \widehat{\sigma}$ *and*

$$\begin{aligned} \sigma' &= \sigma[ref \mapsto d] \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup [\widehat{ref} \mapsto \widehat{d}] \end{aligned}$$

then it follows that $\sigma' \mathcal{R}_{Store} \widehat{\sigma}'$*.*

Proof. From $\widehat{\sigma} \sqcup [\widehat{ref} \mapsto \widehat{d}]$ follows that $\widehat{\sigma}'(\widehat{ref}) = \widehat{\sigma}(\widehat{ref}) \sqcup \widehat{d}$ which by Lemma 2 means that $\sigma(ref) \mathcal{R}_{Store} \widehat{\sigma}'(\widehat{ref})$. By $\sigma \mathcal{R}_{Store} \widehat{\sigma}$ it is clear that

$$\sigma \mathcal{R}_{Store} \widehat{\sigma} \text{ iff } \forall ref \in \mathrm{dom}(\sigma) : \forall \widehat{ref} \in \widehat{\mathbf{Ref}} : ref \mathcal{R}_{Ref} \widehat{ref} \Rightarrow \sigma(ref) \mathcal{R}_D \widehat{\sigma}(\widehat{ref})$$

holds. Both facts considered together show that $\sigma' \mathcal{R}_{Store} \widehat{\sigma}'$ holds. □

The next lemma states that under matching environments the argument evaluation functions yield the correct result:

Lemma 6 (Argument evaluation)**.** *Let* $\beta \in \mathbf{BEnv}$*,* $\widehat{\beta} \in \widehat{\mathbf{BEnv}}$*,* $ve \in \mathbf{VEnv}$*,* $\widehat{ve} \in \widehat{\mathbf{VEnv}}$*,* $\sigma \in \mathbf{Store}$*,* $\widehat{\sigma} \in \widehat{\mathbf{Store}}$*,* $t \in \mathbf{Time}$*, and* $\widehat{t} \in \widehat{\mathbf{Time}}$ *such that* $\beta \mathcal{R} \widehat{\beta}$*,* $ve \mathcal{R} \widehat{ve}$*,* $\sigma \mathcal{R} \widehat{\sigma}$*, and* $t \mathcal{R} \widehat{t}$*. Then:*

$$(\mathcal{A}\, \beta\, ve\, \sigma\, t\, e) \mathcal{R} (\widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}\, e)$$

Proof. Case analysis for e:

1. Evaluating a lambda expression: $e \in$ `Lam`, $e = lam_\tau$

   $\mathcal{A}\, \beta\, ve\, \sigma\, t\, e$ evaluates to the closure (lam_τ, β, t). $\widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t}\, e$ evaluates to the abstract denotable value $\{(lam_\tau, \widehat{\beta}, \widehat{t})\}$ which directly leads to $(lam_\tau, \beta, t) \mathcal{R}_{Proc} (lam_\tau, \widehat{\beta}, \widehat{t})$ because $\beta \mathcal{R} \widehat{\beta}$ and $t \mathcal{R} \widehat{t}$ hold by premise.

2. Evaluating a lexical variable reference: $e \in$ `Var`, $e = v$

   $\mathcal{A}$ looks up the value of v using β and ve: $ve(v, \beta(v))$. $\widehat{\mathcal{A}}$ looks up the values in the abstract counterpart: $\widehat{ve}(v, \widehat{\beta}(v))$. By the premise $\beta \mathcal{R} \widehat{\beta}$ it is clear that $\beta(v) \mathcal{R}_{Time} \widehat{\beta}(v)$. Since $ve \mathcal{R} \widehat{ve}$ it is also clear that for all $t \mathcal{R}_{Time} \widehat{t}$ the abstract variable environment stores the correct abstract values: $ve(v, t) \mathcal{R}_D \widehat{ve}(v, \widehat{t})$.

3. Evaluating a reference to a global variable: $e \in$ `GVar`, $e = g$

   $\mathcal{A}$ finds the global variable g in the store σ: $\sigma(g)$. $\widehat{\mathcal{A}}$ consults the abstract store $\widehat{\sigma}$: $\widehat{\sigma}(g)$. By the premise $\sigma \mathcal{R} \widehat{\sigma}$ the condition $\sigma(g) \mathcal{R} \widehat{\sigma}(g)$ holds.

4. Evaluating a literal: $e \in \texttt{Lit}_{Lang}$, $e = lit$

   The semantics of the intermediate language does not include literals. Therefore the evaluation of literals is shifted to the language-specific evaluation function $\mathcal{K}_{Lang}$ and $\widehat{\mathcal{K}}_{Lang}$. The abstract literal evaluation function, however, must obey $\mathcal{K}_{Lang}\, lit \mathcal{R}_D \widehat{\mathcal{K}}_{Lang}\, lit$.

5. Evaluating a trivial call: $e \in \texttt{TrivCall}$, $e = (prim\ \langle e^u_1, \ldots, e^u_n \rangle)_\tau$

   The primitive operation used in trivial calls are specific to the input language. Thus, they are not considered in the intermediate language and shifted to the evaluation functions $\mathcal{P}_{Lang}$ and $\widehat{\mathcal{P}}_{Lang}$. The definitions of these functions must ensure that

   $$(\mathcal{P}_{Lang}\ \beta\ ve\ \sigma\ t\ prim\ \langle e_1, \ldots, e_n \rangle)\ \mathcal{R}_D\ (\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t'}\ prim_l \langle e_1, \ldots, e_n \rangle)$$

   holds. Note, that trivial calls by definition do not have side effects and do not affect the control flow. Thus, the stated condition is sufficient.

□

The approximation relation $\sqsubseteq$ relates elements from abstract domains. Intuitively, if for an abstract entity $\widehat{\xi}$ (an abstract value, environment, or state) $\xi\ \mathcal{R}\ \widehat{\xi}$ holds and there is an abstract entity $\widehat{\xi'}$ such that $\widehat{\xi} \sqsubseteq \widehat{\xi'}$ then one would expect $\xi\ \mathcal{R}\ \widehat{\xi'}$ to hold as well. The next seven lemmas prove this property for binding environments, procedure values, locations, references, denotable values, variable environments, and stores.

Lemma 7 (Correctness and approximation of binding environments)**.** *Let* $\beta \in$ **BEnv** *and* $\widehat{\beta}, \widehat{\beta'} \in \widehat{\mathbf{BEnv}}$*. Then:*

$$\beta\ \mathcal{R}\ \widehat{\beta} \quad \wedge \quad \widehat{\beta} \sqsubseteq \widehat{\beta'} \qquad \Rightarrow \beta\ \mathcal{R}\ \widehat{\beta'}$$

Proof. By definition, $\widehat{\beta} \sqsubseteq \widehat{\beta'}$ is equivalent to

$$\forall v \in \text{dom}(\widehat{\beta}) : \ \widehat{\beta}(v) \sqsubseteq_{Time} \widehat{\beta'}(v)$$

That is, $\text{dom}(\widehat{\beta}) \subseteq \text{dom}(\widehat{\beta'})$. Recall the premise for time abstractions stated in Section 4.3:

$$t\ \mathcal{R}_{Time}\ \widehat{t}\ \wedge\ \widehat{t} \sqsubseteq \widehat{t'} \quad \Rightarrow \quad t\ \mathcal{R}_{Time}\ \widehat{t'}$$

These two facts considered together give

$$\forall v \in \text{dom}(\widehat{\beta}) : \beta(v)\ \mathcal{R}_{Time}\ \widehat{\beta'}(v)$$

which is the definition of $\beta\ \mathcal{R}_{BEnv}\ \widehat{\beta'}$. □

Lemma 8 (Correctness and approximation of procedure values)**.** *Let* $proc \in$ **Proc** *and* $\widehat{proc}, \widehat{proc'} \in \widehat{\mathbf{Proc}}$*. Then:*

$$proc\ \mathcal{R}\ \widehat{proc} \quad \wedge \quad \widehat{proc} \sqsubseteq \widehat{proc'} \qquad \Rightarrow proc\ \mathcal{R}\ \widehat{proc'}$$

Proof. The interesting cases are:

1. $proc = \texttt{halt}$ and $\widehat{proc} = \texttt{halt}$. From, $\widehat{proc} \sqsubseteq \widehat{proc'}$ it is clear that $\widehat{proc'} = \texttt{halt}$ or $\widehat{proc'} = \top_{Proc}$. For both cases $proc\ \mathcal{R}\ \widehat{proc'}$ holds.

2. $proc \in$ **Clo** and $\widehat{proc} \in \widehat{\mathbf{Clo}}$. Since $proc\ \mathcal{R}\ \widehat{proc}$ the closures are over the same lambda expressions (by definition of $\mathcal{R}_{Proc}$). Then, $\widehat{proc'}$ must be an abstract closure (or $\top_{Proc}$) over the same lambda term since $\sqsubseteq$ on syntactic domains is equality. Let $\widehat{\beta'}$ be the binding environment and the time $\widehat{t'}$ of $\widehat{proc'}$.

   Then, $\mathcal{R}_{Proc}$ considers the components of an abstract closure:

(a) The lambda nodes for $proc$, $\widehat{proc}$, and $\widehat{proc'}$ are identical, as just discussed.

(b) By definition of $\sqsubseteq$ for product lattices and the premise $\widehat{proc'} \sqsubseteq \widehat{proc'}$ $\widehat{\beta} \sqsubseteq \widehat{\beta'}$ follows. From $proc \, \mathcal{R} \, \widehat{proc}$ also $\beta \, \mathcal{R} \, \widehat{\beta}$ follows. This establishes the premises for applying Lemma 7 from which $\beta \; \mathcal{R} \; \widehat{\beta'}$ follows.

(c) By the premises for time abstractions (see Section 4.3), it is clear that $t \; \mathcal{R} \; \widehat{t'}$.

$\square$

Lemma 9 (Correctness and approximation of locations)**.** *Let $loc \in \mathbf{Loc}$ and $\widehat{loc}, \widehat{loc'} \in \widehat{\mathbf{Loc}}$. Then:*

$$loc \; \mathcal{R} \; \widehat{loc} \quad \wedge \quad \widehat{loc} \sqsubseteq \widehat{loc'} \qquad \Rightarrow loc \; \mathcal{R} \; \widehat{loc'}$$

Proof. The interesting cases are:

1. $loc \in \mathbf{GLoc}$. So that $loc \; \mathcal{R} \; \widehat{loc}$ holds, $\widehat{loc}$ must be a global location $\widehat{loc} \in \widehat{\mathbf{GLoc}}$. Then, loc and $\widehat{loc}$ go back to global variables — a syntactic domain. Here, $\sqsubseteq$ is equality and from $\widehat{loc} \sqsubseteq \widehat{loc'}$ it follows $\widehat{loc} = \widehat{loc'}$.

2. $loc \in \mathbf{HLoc}$. Then, $\widehat{loc} \in \widehat{\mathbf{HLoc}}$. By definition of $\mathcal{R}_{HLoc}$ the call constituents of loc and $\widehat{loc}$ must be identical. On the syntactic domain `Call` $\sqsubseteq$ is defined as equality and hence $\widehat{loc}$ and $\widehat{loc'}$ must contain the same call. By premise for time abstraction (see Section 4.3) the time constituent obeys $t \; \mathcal{R} \; \widehat{t'}$. Considered together, this yields $loc \; \mathcal{R} \; \widehat{loc'}$.

$\square$

Lemma 10 (Correctness and approximation of references)**.** *Let $ref \in \mathbf{Ref}$ and $\widehat{ref}, \widehat{ref'} \in \widehat{\mathbf{Ref}}$. Then:*

$$ref \; \mathcal{R} \; \widehat{ref} \quad \wedge \quad \widehat{ref} \sqsubseteq \widehat{ref'} \qquad \Rightarrow ref \; \mathcal{R} \; \widehat{ref'}$$

Proof. $\mathcal{R}_{Ref}$ considers the constituents of references: A location and a selector from $\texttt{Lit}_{Lang}$. Since literals are a syntactic domain, $\sqsubseteq$ on $\texttt{Lit}_{Lang}$ is equality and therefore the literal in ref, $\widehat{ref}$, and $\widehat{ref'}$ are identical. From $\widehat{ref} \sqsubseteq \widehat{ref'}$ and the definition of $\sqsubseteq$ on product lattices, for the $\widehat{loc}$ in $\widehat{ref}$ and the $\widehat{loc'}$ in $\widehat{ref'}$ $\widehat{ref} \sqsubseteq \widehat{ref'}$ follows. This establishes the premises for Lemma 9 and both facts considered together yields $ref \; \mathcal{R} \; \widehat{ref'}$. $\square$

Lemma 11 (Correctness and approximation of denotable values)**.** *Let $d \in \mathbf{D}$ and $\widehat{d}, \widehat{d'} \in \widehat{\mathbf{D}}$. Then:*

$$d \; \mathcal{R} \; \widehat{d} \quad \wedge \quad \widehat{d} \sqsubseteq \widehat{d'} \qquad \Rightarrow d \; \mathcal{R} \; \widehat{d'}$$

Proof. From the premise $d \, \mathcal{R} \, \widehat{d}$ and the definition of $\mathcal{R}_D$ it is immediately clear that $\exists \widehat{v} \in \widehat{d} : d \; \mathcal{R} \; \widehat{v}$. So, the obligation is to prove that this value also exists in $\widehat{d'}$. $\widehat{\mathbf{D}}$ is a power lattice and therefore from the premise $\widehat{d} \sqsubseteq \widehat{d'}$ it is clear that

$$\forall \widehat{b_1} \in \widehat{d} : \; \exists \widehat{b_2} \in \widehat{d'} : \; \widehat{b_1} \sqsubseteq \widehat{b_2}$$

So, there exists an abstract value $\widehat{v} \in \widehat{d}$ for which $d \ \mathcal{R} \ \widehat{v}$ and there exists a value in $\widehat{d'}$ such that $\widehat{v} \sqsubseteq \widehat{v'}$. By Lemmas 8, 9, and 10 it is clear that $v \ \mathcal{R} \ \widehat{v'}$ for v being a procedure value, location, or reference. For language specific basic values ($v \in \mathbf{Bas}_{Lang}$) this is a premise. ☐

Lemma 12 (Correctness and approximation of variable environments)**.** *Let* $ve \in \mathbf{VEnv}$ *and* $\widehat{ve}, \widehat{ve'} \in \widehat{\mathbf{VEnv}}$*. Then:*

$$ve \ \mathcal{R} \ \widehat{ve} \quad \wedge \quad \widehat{ve} \sqsubseteq \widehat{ve'} \qquad \Rightarrow ve \ \mathcal{R} \ \widehat{ve'}$$

Proof. From the premise $ve \ \mathcal{R} \ \widehat{ve}$ it follows that

$$\forall (v,t) \in dom(ve): \ \forall \widehat{t} \in \widehat{\mathbf{Time}} : t \ \mathcal{R} \ \widehat{t} \Rightarrow ve(v,t) \ \mathcal{R}_D \ \widehat{ve}(v,\widehat{t})$$

Thus, the proof obligation is to show that for all points in time $\widehat{t} \in \widehat{\mathbf{Time}}$ such that $t \ \mathcal{R} \ \widehat{t}$ and all variables v there $\widehat{ve}(v,\widehat{t}) \ \mathcal{R}_D \ \widehat{ve'}(v,\widehat{t})$ holds. From the definition of $\sqsubseteq$ on function lattices and the premise $\widehat{ve} \sqsubseteq \widehat{ve'}$ it is clear that

$$\forall (v,\widehat{t}) \in \mathrm{dom}(\widehat{ve}) : \widehat{ve}(v,\widehat{t}) \sqsubseteq_D \widehat{ve'}(v,\widehat{t})$$

This, however, is the premise for applying Lemma 11 and hence $\widehat{ve} \ \mathcal{R} \ \widehat{ve'}$ follows. ☐

Lemma 13 (Correctness and approximation of stores)**.** *Let* $\sigma \in \mathbf{Store}$ *and* $\widehat{\sigma}, \widehat{\sigma'} \in \widehat{\mathbf{Store}}$*. Then:*

$$\sigma \ \mathcal{R} \ \widehat{\sigma} \quad \wedge \quad \widehat{\sigma} \sqsubseteq \widehat{\sigma'} \qquad \Rightarrow \sigma \ \mathcal{R} \ \widehat{\sigma'}$$

Proof. The proof for this lemma can be adapted *mutatis mutandis* from Lemma 12. ☐

The preceding seven lemma faciliate formulating the theorem that connects the semantics correctness $\mathcal{R}$ with the approximation relation $\sqsubseteq$ for states. This theorem is necessary for proving the central semantic correctness theorem.

Theorem 2 (Correctness and approximation of states)**.** *Let* $\varsigma \in \mathbf{State}$ *and* $\widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\mathbf{State}}$.

$$\varsigma \ \mathcal{R} \ \widehat{\varsigma} \quad \wedge \quad \widehat{\varsigma} \sqsubseteq \widehat{\varsigma}' \qquad \Rightarrow \varsigma \ \mathcal{R} \ \widehat{\varsigma}'$$

Proof. $\widehat{\mathbf{State}}$ is a sum lattice and by definition of $\sqsubseteq$ and definition of $\mathcal{R}_{State}$, $\widehat{\varsigma} \sqsubseteq \widehat{\varsigma}'$ only holds if $\widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\mathbf{Eval}}$ or $\widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\mathbf{Apply}}$. Thus, the proof considers the following cases:

1. $\varsigma \in \mathbf{Eval}$ and $\widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\mathbf{Eval}}$. That is:

$$\begin{aligned} \varsigma &= (call_\tau, \beta, ve, \sigma, t) \\ \widehat{\varsigma} &= (call_{\tau'}, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \\ \widehat{\varsigma}' &= (call_{\tau''}, \widehat{\beta}', \widehat{ve}', \widehat{\sigma}', \widehat{t}') \end{aligned}$$

The correctness relation $\mathcal{R}_{State}$ requires that all constituents of an *eval* state obey their respective correctness relations. From $\varsigma \ \mathcal{R} \ \widehat{\varsigma}$ follows $call_\tau = call_{\tau'}$. From $\widehat{\varsigma} \sqsubseteq \widehat{\varsigma}'$ follows that the calls from $\widehat{\varsigma}$ and $\widehat{\varsigma}'$ are

related by $\sqsubseteq$. `Call` is a syntactic domain and hence $\sqsubseteq$ for calls is equality. Thus, the call constituents of $\widehat{\varsigma}$ and $\widehat{\varsigma}'$ are identical.

From $\varsigma\ \mathcal{R}\ \widehat{\varsigma}$ it follows that $\beta\ \mathcal{R}_{BEnv}\ \widehat{\beta}$, $ve\ \mathcal{R}_{VEnv}\ \widehat{ve}$, and $\sigma\ \mathcal{R}_{Store}\ \widehat{\sigma}$. From $\widehat{\varsigma} \sqsubseteq \widehat{\varsigma}'$ it follows that $\widehat{\beta} \sqsubseteq \widehat{\beta}'$, $\widehat{ve} \sqsubseteq \widehat{ve}'$, and $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$. This establishes the premises of Lemma 7, 12, and 13.

For the time constituent of the *eval* states $t\ \mathcal{R}_{Time}\ \widehat{t}'$ follows by premise (see Section 4.3).

2. $\varsigma \in \mathbf{Apply}$ and $\widehat{\varsigma}, \widehat{\varsigma}' \in \widehat{\mathbf{Apply}}$. That is,

$$\begin{aligned}
\varsigma &= (\{proc\}, d^*, c^*, ve, \sigma, t)\\
\widehat{\varsigma} &= (\{\widehat{proc}\}, \widehat{d}^*, \widehat{c}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t})\\
\widehat{\varsigma}' &= (\{\widehat{proc}'\}, \widehat{d}'^*, \widehat{c}'^*, \widehat{ve}', \widehat{\sigma}', \widehat{t}')
\end{aligned}$$

The correctness relation $\mathcal{R}_{State}$ requires that all constituents of an *apply* state obey the correctness relations. From $\varsigma\ \mathcal{R}\ \widehat{\varsigma}$ it follows that $proc\ \mathcal{R}_{Proc}\ \widehat{proc}$, $ve\ \mathcal{R}\ \widehat{ve}$, and $\sigma\ \mathcal{R}_{Store}\ \widehat{\sigma}$. From the premise $\widehat{\varsigma} \sqsubseteq \widehat{\varsigma}'$ follows $\widehat{proc} \sqsubseteq \widehat{proc}'$, $\widehat{ve} \sqsubseteq \widehat{ve}'$, and $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$. This establishes the premises necessary for applying Lemma 8, 12, and 13.

For the time constituent of the *apply* states $t\ \mathcal{R}_{Time}\ \widehat{t}'$ follows by premise (see Section 4.3).

For vectors $\mathcal{R}$ works element-wise: $d^*\ \mathcal{R}_{D^*}\ \widehat{d}^*$ holds if for all elements d_i from vector d^* $d_i\ \mathcal{R}_D\ \widehat{d}_i$ holds. From the premise $\widehat{d}^* \sqsubseteq \widehat{d}'^*$ follows that $\widehat{d}_i \sqsubseteq \widehat{d}'_i$ for all elements $\widehat{d}_i$ from $\widehat{d}^*$. That is, the premises for applying Lemma 11 hold for each element of the argument vectors $\widehat{c}^*$ and $\widehat{d}^*$.

□

Equipped with this theorem and these lemma, it is now possible to formulate and prove the main correctness theorem:

Theorem 3 (Semantic correctness)**.** *For $\varsigma \in \mathcal{V}(pr)$ there exists $\widehat{\varsigma} \in \widehat{\mathcal{V}}(pr)$ such that $\varsigma\ \mathcal{R}\ \widehat{\varsigma}$.*

Proof. The theorem follows by induction over the state transitions.

The base case considers the initial states: $\varsigma_i = \mathcal{I}(pr)$ and $\widehat{\varsigma}_i = \widehat{\mathcal{I}}(pr)$. Thus, $\mathcal{V}(pr) = \{\varsigma_i\}$ and $\widehat{\mathcal{V}}(pr) = \{\widehat{\varsigma}_i\}$. By inspection of the definitions for $\mathcal{I}$ and $\widehat{\mathcal{I}}$ it is clear that $\varsigma_i\ \mathcal{R}\ \widehat{\varsigma}_i$.

The obligation is to prove that if $\varsigma \twoheadrightarrow \varsigma'$ an abstract state $\widehat{\varsigma}'$ with $\varsigma'\ \mathcal{R}\ \widehat{\varsigma}'$ exists. There are two cases to consider: First, $\widehat{\mathcal{V}}(pr)$ may contain an abstract state $\widehat{\varsigma}''$ such that $\widehat{\varsigma}' \sqsubseteq \widehat{\varsigma}''$. Then, theorem 2 applies and $\varsigma\ \mathcal{R}\ \widehat{\varsigma}''$ follows. In the second case there exists no element $\widehat{\varsigma}''$ in $\widehat{\mathcal{V}}(pr)$ that $\widehat{\varsigma}' \sqsubseteq \widehat{\varsigma}''$. That is, the abstract successor state $\widehat{\varsigma}'$ will be freshly added to $\widehat{\mathcal{V}}(pr)$. Then for $\widehat{\varsigma}'$ the condition $\varsigma\ \mathcal{R}\ \widehat{\varsigma}'$ must hold. This obligation follows by inspecting the cases and transition rules.

The cases are:

1. Let $\varsigma = ((prim_p\ \langle c, f, a_1, \ldots, a_n\rangle)_\tau, \beta, ve, \sigma, t)$ and $prim_p \in \texttt{Prim}_{PCall} \setminus \{\texttt{letrec1}, \texttt{letrec2}\}$. By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state $\widehat{\varsigma} = ((prim_p\ \langle c, f, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})$ such that $\varsigma\ \mathcal{R}\ \widehat{\varsigma}$. For ς

the transition rule PCallEval applies: $\varsigma \rightarrow \varsigma'$. For $\widehat{\varsigma}$ the transition rule $\widehat{\text{PCallEval}}$ applies: $\widehat{\varsigma \rightarrow \varsigma'}$. The obligation is to prove $\varsigma' \; \mathcal{R} \; \widehat{\varsigma}'$.

Both transition rules advance time using *tick* and $\widehat{tick}$ respectively. Recall the prerequisite for the time abstractions stated in Section 4.3:

$$t \; \mathcal{R}_{Time} \; \widehat{t} \Rightarrow tick(t) \; \mathcal{R}_{Time} \; \widehat{tick}(\widehat{t})$$

Thus, for the points in time t' and $\widehat{t'}$ in the successor states the condition $t' \; \mathcal{R}_{Time} \; \widehat{t'}$ holds.

PCallEval and $\widehat{\text{PCallEval}}$ evaluate the operator expression f and the continuation expression c:

$$\begin{aligned} proc &= \mathcal{A} \, \beta \; ve \; \sigma \; t' \; f \\ \widehat{proc} &= \widehat{\mathcal{A}} \, \widehat{\beta} \; \widehat{ve} \; \widehat{\sigma} \; \widehat{t'} \; f \\[1em] c' &= \mathcal{A} \, \beta \; ve \; \sigma \; t' \; c \\ \widehat{c'} &= \widehat{\mathcal{A}} \, \widehat{\beta} \; \widehat{ve} \; \widehat{\sigma} \; \widehat{t'} \; c \end{aligned}$$

By lemma 6 it is clear, that $proc \; \mathcal{R} \; \widehat{proc}$ and $c' \; \mathcal{R} \; \widehat{c'}$ hold.

Furthermore, both transition rules evaluate the call arguments a_i using $\mathcal{A}$ and $\widehat{\mathcal{A}}$:

$$\begin{aligned} d_i &= \mathcal{A} \, \beta \; ve \; \sigma \; t' \; a_i \\ \widehat{d}_i &= \widehat{\mathcal{A}} \, \widehat{\beta} \; \widehat{ve} \; \widehat{\sigma} \; \widehat{t'} \; a_i \end{aligned}$$

Again, by lemma 6, $d_i \; \mathcal{R} \; \widehat{d}_i$.

In summary, $t' \, \mathcal{R}_{Time} \, \widehat{t'}$, $proc \, \mathcal{R} \, \widehat{proc}$, $c' \, \mathcal{R} \, \widehat{c'}$, and $d_i \, \mathcal{R} \, \widehat{d}_i$ hold. Considered together with the premises $ve \; \mathcal{R} \; \widehat{ve}$ and $\beta \; \mathcal{R} \; \widehat{\beta}$ and the continuation of $\mathcal{R}$ on vectors (see Section 4.3), all constituents of the apply states resulting from the transition are related. Thus,

$$(proc, \langle c' \rangle, d^*, ve, \sigma, t') \; \mathcal{R} \; (\{\widehat{p}_i\}, \langle \widehat{c'} \rangle, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t'})$$

with $\widehat{p}_i \in \widehat{proc}$ holds. Consequently, $\varsigma' \; \mathcal{R} \; \widehat{\varsigma}'$.

2. Let $\varsigma = ((prim_c \; \langle c, a_1, \ldots, a_n \rangle)_\tau, \beta, ve, \sigma, t)$. The transition rules CCallEval and $\widehat{\text{CCallEval}}$ apply. The proof for this case can be adapted *mutatis mutandis* from case 1.

3. Let $\varsigma = ((prim_l \; \langle c, a_1, \ldots, a_n \rangle)_\tau, \beta, ve, \sigma, t)$. The transition rules PrimCallEval and $\widehat{\text{PrimCallEval}}$ apply. The proof for this case can be adapted *mutatis mutandis* from case 1.

4. Let $\varsigma = ((\texttt{test} \; \langle c_0, c_1, a \rangle)_\tau, \beta, ve, \sigma, t)$. By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state $\widehat{\varsigma} = ((\texttt{test} \; \langle c_0, c_1, a \rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})$. The transition rules TestEval and $\widehat{\text{TestEval}}$ apply. Thus, $\varsigma \rightarrow \varsigma'$ and $\widehat{\varsigma \rightarrow} \{\widehat{\varsigma}_0, \widehat{\varsigma}_1\}$. The obligation is to show that $\varsigma' \; \mathcal{R} \; \widehat{\varsigma}_0 \vee \varsigma' \; \mathcal{R} \; \widehat{\varsigma}_1$ holds.

   The concrete machine goes into the following *apply* state:

$$\varsigma' = (c', \langle\rangle, \langle\rangle, ve, \sigma, \widehat{t'})$$

c' is the result of the evaluation of c_0 or c_1 and depends on the value of a:

$$\begin{aligned} r &= \mathcal{A}\,\beta\;ve\;\sigma\;t'\;a \\ c' &= \begin{cases} \mathcal{A}\,\beta\;ve\;\sigma\;t'\;c_0 & \text{iff } \mathit{trueish?}(r) \\ \mathcal{A}\,\beta\;ve\;\sigma\;t'\;c_1 & \text{otherwise} \end{cases} \end{aligned}$$

The abstract machine creates two *apply* states:

$$\begin{aligned} \widehat{\varsigma_0} &= (\widehat{b_0}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'}) \\ \widehat{\varsigma_1} &= (\widehat{b_1}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'}) \end{aligned}$$

where $\widehat{b_i} = \widehat{\mathcal{A}}\,\widehat{\beta}\;\widehat{ve}\;\widehat{\sigma}\;\widehat{t'}\;c_i$.

Note that for the binding environments, variable environments, and the continuation and user-world arguments the following holds: $\beta\,\mathcal{R}\,\widehat{\beta}$, $ve\,\mathcal{R}\,\widehat{ve}$, $\langle\rangle\;\mathcal{R}\;\langle\rangle$. Furthermore, both transistion rules advance the time. Following the argumentation seen in case 1, for the points in time t' in the successor states $t'\;\mathcal{R}\;\widehat{t'}$ holds.

There are two cases to distinguish:

(a) Let $c' = \mathcal{A}\,\beta\;ve\;\sigma\;t'\;c_0$. In this case $c'\;\mathcal{R}\;\widehat{b_0}$ holds and consequently also $\varsigma'\;\mathcal{R}\;\widehat{\varsigma_0}$.

(b) Let $c' = \mathcal{A}\,\beta\;ve\;\sigma\;t'\;c_1$. In this case $c'\;\mathcal{R}\;\widehat{b_1}$ holds and consequently also $\varsigma'\;\mathcal{R}\;\widehat{\varsigma_1}$.

Note that either case 4a or case 4b applies. Considered together, this means $\varsigma'\;\mathcal{R}\;\widehat{\varsigma_0} \vee \varsigma'\;\mathcal{R}\;\widehat{\varsigma_1}$.

5. This case considers *eval* states with calls to `letrec` such as the following:

$$\varsigma = \begin{array}{l} (\texttt{(letrec1}\;\langle(\lambda\;(v_1^c \ldots v_n^c) \\ \qquad \texttt{(letrec2}\;\langle l^c, e_1^u \ldots e_n^u\rangle)_\tau\rangle)_{\tau'})_{\tau''}, \beta, ve, \sigma, t) \end{array}$$

By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state

$$\widehat{\varsigma} = \begin{array}{l} (\texttt{(letrec1}\;\langle(\lambda\;(v_1^c \ldots v_n^c) \\ \qquad \texttt{(letrec2}\;\langle l^c, e_1^u \ldots e_n^u\rangle)_\tau\rangle)_{\tau'})_{\tau''}, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \end{array}$$

such that $\varsigma\;\mathcal{R}\;\widehat{\varsigma}$. The transition rules LetrecEval and $\widehat{\text{LetrecEval}}$ apply:

$$\begin{array}{ccc} \varsigma & \rightarrow & \varsigma' \\ \widehat{\varsigma} & \widehat{\rightarrow} & \widehat{\varsigma'} \end{array}$$

The obligation is to prove $\varsigma'\;\mathcal{R}\;\widehat{\varsigma'}$.

Both transition rules advance the time to t' and $\widehat{t'}$ and by the premises for time abstractions $t'\;\mathcal{R}\;\widehat{t'}$ holds.

Like transitions for *apply* states, LetrecEval and $\widehat{\text{LetrecEval}}$ extend the binding and variable environments and bind variables v_i to the values on right-hand side r_i and $\widehat{r_i}$ respectively:

$$\begin{aligned} \beta' &= \beta[v_i \mapsto t'] \\ ve' &= ve[(v_i^c, t') \mapsto r_i] \\ r_i &= \mathcal{A}\,\beta'\,ve\,\sigma\,t'\,e_i^u \end{aligned}$$

$$\begin{aligned} \widehat{\beta}' &= \widehat{\beta}[v_i \mapsto \widehat{t'}] \\ \widehat{ve}' &= \widehat{ve} \sqcup [(v_i^c, \widehat{t'}) \mapsto \widehat{r_i}] \\ \widehat{r_i} &= \widehat{\mathcal{A}}\,\widehat{\beta}'\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,e_i^u \end{aligned}$$

By Lemma 4 $\beta' \;\mathcal{R}\; \widehat{\beta}'$ is true. Because $t' \;\mathcal{R}\; \widehat{t'}$ and $ve \;\mathcal{R}\; \widehat{ve}$ and Lemma 6, $r_i \;\mathcal{R}\; \widehat{r_i}$ holds for all i. Consequently, by Lemma 3 $ve' \;\mathcal{R}\; \widehat{ve}'$ follows. These facts considered together give:

$$(call, \beta', ve', \sigma, t') \;\mathcal{R}\; (call, \widehat{\beta}', \widehat{ve}', \widehat{\sigma}, \widehat{t'}) \quad \Leftrightarrow \quad \varsigma' \;\mathcal{R}\; \widehat{\varsigma}'$$

6. Let $\varsigma = (\texttt{(global-ref}\ \langle c, g\rangle\texttt{)}_\tau, \beta, ve, \sigma, t)$. By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state

   $$\widehat{\varsigma} = (\texttt{(global-ref}\ \langle c, g\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})$$

   such that $\varsigma \;\mathcal{R}\; \widehat{\varsigma}$. For ς the transition rule PGlobalRef applies and for $\widehat{\varsigma}$ the rule $\widehat{\text{PGlobalRef}}$ applies. Thus, the resulting states are *apply* states.

   Both rules advance the time using *tick* and $\widehat{tick}$ and by the argumentation laid out for case 1, $t' \;\mathcal{R}\; \widehat{t'}$ holds. By Lemma 6 it is clear that the continuation argument c evaluates to the correct procedure values. Since $\sigma \;\mathcal{R}\; \widehat{\sigma}$ by induction hypothesis, $\sigma(g) \;\mathcal{R}\; \widehat{\sigma}(g)$ follows. That is, the arguments of the abstract *apply* state simulate the arguments of the concrete *apply* state. These facts considered together give $\varsigma' \;\mathcal{R}\; \widehat{\varsigma}'$.

7. Let $\varsigma = (\texttt{(global-set!}\ \langle c, g, a\rangle\texttt{)}_\tau, \beta, ve, \sigma, t)$. By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state

   $$\widehat{\varsigma} = (\texttt{(global-set!}\ \langle c, g, a\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})$$

   such that $\varsigma \;\mathcal{R}\; \widehat{\varsigma}$. The obligation is to prove $\varsigma' \;\mathcal{R}\; \widehat{\varsigma}'$. Both rules advance time and by premise $t' \;\mathcal{R}\; \widehat{t'}$ holds. Both transition rules evaluate the continuation argument c and the expression a:

   $$\begin{aligned} proc &= \mathcal{A}\,\beta\,ve\,\sigma\,t'\,c \\ \widehat{proc} &= \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \end{aligned}$$

   $$\begin{aligned} d &= \mathcal{A}\,\beta\,ve\,\sigma\,t'\,a \\ \widehat{d} &= \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,a \end{aligned}$$

   By Lemma 6 it is clear that $proc \;\mathcal{R}\; \widehat{proc}$ and $d \;\mathcal{R}\; \widehat{d}$. Both rules update the store:

   $$\begin{aligned} \sigma' &= \sigma[(g, \top) \mapsto d] \\ \widehat{\sigma}' &= \widehat{\sigma} \sqcup [(g, \top) \mapsto \widehat{d}] \end{aligned}$$

Since $d \ \mathcal{R} \ \widehat{d}$ and $\sigma \ \mathcal{R} \ \widehat{\sigma}$ hold, the premises for Lemma 5 are given and $\sigma' \ \mathcal{R} \ \widehat{\sigma}'$ holds. That is, all constituents of the resulting apply states are related by $\mathcal{R}$ and hence $\varsigma' \ \mathcal{R} \ \widehat{\varsigma}'$ follows.

8. Let $\varsigma = (((\lambda\, (p_1 \dots p_n)\ call)_\tau, \beta, t_b), c^*, d^*, ve, \sigma, t)$. By induction hypothesis, $\widehat{\mathcal{V}}(pr)$ contains an abstract state

$$\widehat{\varsigma} = (\{((\lambda\ (p_1 \dots p_n)\ call)_\tau, \widehat{\beta}, \widehat{t_b})\}, \widehat{c}^*, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t})$$

such that $\varsigma \ \mathcal{R} \ \widehat{\varsigma}$. The obligation to show is that for $\varsigma \rightharpoondown \varsigma'$ and $\widehat{\varsigma \rightharpoondown \varsigma'}$ the condition $\varsigma' \ \mathcal{R} \ \widehat{\varsigma}'$ holds.

The transition rules ApplyClos and $\widehat{\text{ApplyClos}}$ apply. Both rules extend the binding and variable environment for the t and $\widehat{t}$ and variables p_i:

$$\begin{aligned}
\beta' &= \beta[p_i \mapsto t] \\
ve' &= ve[(p_i, t) \mapsto (c^* \S d^*)_i] \\
\\
\widehat{\beta}' &= \widehat{\beta}[p_i \mapsto \widehat{t}] \\
\widehat{ve}' &= \widehat{ve} \sqcup [(p_i, \widehat{t}) \mapsto (\widehat{c}^* \S \widehat{d}^*)_i]
\end{aligned}$$

Since the prerequisites for Lemma 3 and Lemma 4 are given, the following holds for the updated environments: $\beta' \ \mathcal{R}_{BEnv} \ \widehat{\beta}'$ and $ve' \ \mathcal{R}_{VEnv} \ \widehat{ve}'$. Consequently, $\varsigma' \ \mathcal{R} \ \widehat{\varsigma}'$ follows.

□

Appendix C

PreScheme Primops

Realistic PreScheme programs such as the Scheme 48 virtual machine use almost all available primops. Hence, the flow analysis needs abstract evaluation functions for these primops to approximate their behavior.

C.1 Primops with one return value

The following table documents the PreScheme primops that simply compute and return a single value. The table sorts the primops by their return-value type. Most of the primops can occur in trivial and in regular calls.

For trivial calls, the following formula for $\widehat{\mathcal{P}}_{Lang}$ serves as a template. $\mathcal{N}$ denotes the primop name from the table, and $\widehat{\mathcal{R}}$ the corresponding return value:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ \mathcal{N}\langle a_0, \ldots, a_n\rangle = \{\widehat{\mathcal{R}}\}$$

If a primop from the table occurs in a regular call, i. e. in a lambda body, this transition rule applies:

$$\frac{}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{c_i}\}, \langle\rangle, \langle\{\widehat{\mathcal{R}}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{c_i} & \in \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t'}\ c \end{cases}$$

Name $\mathcal{N}$	**Argument Types**	**Description**
Primops returning the abstract integer value $\widehat{\mathcal{R}} = \top_{Int}$:		
`*`	`int` $\times$ `int`	integer multiplication
`+`	`int` $\times$ `int`	integer addition
`-`	`int` $\times$ `int`	integer subtraction
`address->integer`	`addr`	memory address as integer number
`ashl`	`int` $\times$ `int`	arithmetic shift left
`ashr`	`int` $\times$ `int`	arithmetic shift right
`bitwise-and`	`int` $\times$ `int`	bitwise and

Name $\mathcal{N}$	**Argument Types**	**Description**
bitwise-ior	int $\times$ int	bitwise or
bitwise-xor	int $\times$ int	bitwise exclusive or
byte-ref	addr	read byte from memory address
char->ascii	char	convert character to ASCII code
close-input-port	inport	close input port
close-output-port	outport	close output port
force-output	outport	flush port buffers
lshr	int $\times$ int	logical shift right
quotient	int $\times$ int	quotient of integer division
remainder	int $\times$ int	remainder of integer division
string-length	string	length of a string
word-ref	addr	read word from memory address
write-block	outport $\times$ addr $\times$ int	write memory contents to port
write-char	char $\times$ outport	output a char to port
write-integer	int $\times$ outport	output an integer to port
write-string	string $\times$ outport	output a string to port
Primops returning the abstract float value $\widehat{\mathcal{R}} = \top_{Float}$:		
fl*	float $\times$ float	multiplication for floating point numbers
fl+	float $\times$ float	addition for floating point numbers
fl-	float $\times$ float	subtraction for floating point numbers
fl/	float $\times$ float	division for floating point numbers
flonum-ref	addr	read floating point from memory address
Primops returning the abstract char value $\widehat{\mathcal{R}} = \top_{Char}$:		
ascii->char	int	return char for ASCII code
string-ref	string $\times$ int	return a char from string by index
Primops returning the abstract boolean value $\widehat{\mathcal{R}} = \top_{Bool}$:		
<	int $\times$ int	integer comparison
=	int $\times$ int	integer equality
address<	addr $\times$ addr	address comparison
address=	addr $\times$ addr	address equality
char<?	char $\times$ char	character comparison
char=?	char $\times$ char	character equality
eq?	any $\times$ any	pointer equality
fl<	float $\times$ float	floating point comparison
fl=	float $\times$ float	floating point equality
memory-equal?	addr $\times$ addr $\times$ int	Compare bytes strings in memory
null-pointer?	any	check for null pointer

Name $\mathcal{N}$	Argument Types	Description
Primops returning the abstract null value $\widehat{\mathcal{R}} = \widehat{null}$:		
`null-pointer`		Return null value
Primops returning the abstract input port value $\widehat{\mathcal{R}} = \widehat{input}$:		
`stdin`		return standard input port
Primops returning the abstract output port value $\widehat{\mathcal{R}} = \widehat{output}$:		
`stderr`		return standard error port
`stdout`		return standard output port
Primops returning the abstract string value $\widehat{\mathcal{R}} = \widehat{string}$:		
`error-string`	`string` $\times$ `outport`	output error message to `stderr`
`make-string`	`int`	allocate new string
Primops returning the abstract unit value $\widehat{\mathcal{R}} = \widehat{unit}$:		
`abort`		stop the program
`copy-memory`	`addr` $\times$ `addr` $\times$ `int`	block copy memory
`deallocate-memory`	`addr`	free memory
`deallocate`	`addr`	free memory
`string-set!`	`string` $\times$ `int` $\times$ `char`	set character in string
`unspecific`		create the unspecific value
Primops returning the abstract address value $\widehat{\mathcal{R}} = \widehat{addr}$:		
`address+`	`addr` $\times$ `addr`	add addresses
`address-difference`	`addr` $\times$ `addr`	subtract addresses
`allocate-memory`	`int`	allocate fresh memory
`integer->address`	`int`	convert integer to address

C.2 Primops for compound values

PreScheme has two compound value types: records and vectors. Section 4.4.1 explains the representation of these values in the store. This section summarizes the equations from Section 4.4.1.

Vectors The primop `vector-ref` retrieves an element from the vector and appears in context of a trivial call:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ \texttt{vector-ref}\langle v, i\rangle = \bigsqcup_{((\tau,\widehat{t'}),s)\in\widehat{vref}} \widehat{\sigma}(((\tau,\widehat{t'}),\texttt{elem}))$$
$$\text{where } \widehat{v} = \widehat{\mathcal{A}}\,\widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ v$$
$$\widehat{vref} = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\widehat{j}, \widehat{ref}) \in \widehat{\sigma}(\widehat{u})\}$$

The following transition rules specify the creation and mutation of vectors:

$$\widehat{\varsigma} = (\texttt{(make-vector}\ \langle c, l, i\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}_v\}\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ \widehat{l} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ l \\ \widehat{ref}_v & = ((\tau, \widehat{t'}), \texttt{vector}) \\ \widehat{ref}_e & = ((\tau, \widehat{t'}), \texttt{elem}) \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup [\widehat{ref}_v \mapsto \{(\widehat{l}, \widehat{ref}_e)\},\ \widehat{ref}_e \mapsto \{\widehat{null}\}] \end{cases}$$

$$\widehat{\varsigma} = (\texttt{(vector-set!}\ \langle c, v, i, n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ \widehat{vals} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ v \\ \widehat{new} & = \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ n \\ \widehat{vref} & = \{\widehat{u} : \widehat{u} \in \widehat{vals}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\widehat{j}, \widehat{ref}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \bigsqcup_{((\tau_i, \widehat{t}_i), s) \in \widehat{vref}} [((\tau_i, \widehat{t}_i), \texttt{elem}) \mapsto \widehat{new}] \end{cases}$$

Records Three primops deal with records in PreScheme: `make-record` creates a new instance of a record, `record-set!` updates a field, and `record-ref` returns the value of a given field. Here is specification for these primops (taken from Section 4.4.1):

$$\widehat{\varsigma} = (\texttt{(make-record}\ \langle c, t_{Sym}\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{res}\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\ c \\ \widehat{res} & = \{\widehat{ref}, \widehat{null}\} \\ \widehat{ref} & = ((\tau, \widehat{t'}), \texttt{record}) \\ \widehat{f} & = \bigsqcup_{fn \in rfields(t_{Sym})} [fn \mapsto ((\tau, \widehat{t'}), fn)] \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup [\widehat{ref} \mapsto \{(t_{Sym}, \widehat{f})\}] \sqcup \bigsqcup_{\widehat{r} \in \mathrm{rng}(\widehat{f})} [\widehat{r} \mapsto \bot_{\widehat{\mathbf{D}}}] \end{cases}$$

$$\overline{\widehat{\varsigma} = (\texttt{(record-set!}\ \langle c, r, t_{Sym}, f_{Sym}, n\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{r} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,r \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,n \\ \widehat{ref} & = \{\widehat{u}:\ \widehat{u} \in \widehat{r}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (t_{Sym}, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \displaystyle\bigsqcup_{((\tau_i, \widehat{t_i}), s) \in \widehat{ref}} \left[((\tau_i, \widehat{t_i}), f_{Sym}) \mapsto \widehat{new}\right] \end{cases}$$

The primop `record-ref` occurs in trivial calls and therefore is a case of $\widehat{\mathcal{P}}_{Lang}$ instead of a transition rule:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\, \texttt{record-ref}\langle r, t_{Sym}, f_{Sym}\rangle = \bigsqcup_{((\tau, \widehat{t'}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau, \widehat{t'}), f_{Sym}))$$
$$\text{where}\ \widehat{r} = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}\,r$$
$$\widehat{ref} = \{\widehat{u}:\ \widehat{u} \in \widehat{r}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (t_{Sym}, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\}$$

C.3 Primops with multiple return values

The primops `open-input-file` and `open-output-file` open a file specified by a file name for read and write access, respectively. Both primops return a port and an integer indicating the status of the I/O operation:

$$\frac{prim_l = \texttt{open-input-file}}{\widehat{\varsigma} = (\texttt{(}prim_l\ \langle c, a\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \widehat{\mathbf{r}}, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{\mathbf{r}} & = \langle\{\widehat{inport}\}, \{\top_{Int}\}\rangle \end{cases}$$

$$\frac{prim_l = \texttt{open-output-file}}{\widehat{\varsigma} = (\texttt{(}prim_l\ \langle c, a\rangle\texttt{)}_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \widehat{\mathbf{r}}, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{\mathbf{r}} & = \langle\{\widehat{outport}\}, \{\top_{Int}\rangle\} \end{cases}$$

`Read-char` reads and consumes a single character from an input port. `Peek-char` also reads a single character but does not consume the character read. Both primops return the character read, a boolean value that is true if the the I/O operation has reached the end of the file, and an integer indicating the I/O status:

$$\frac{prim_l \in \{\texttt{read-char}, \texttt{peek-char}\}}{\widehat{\varsigma} = ((prim_l\ \langle c, a\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \widehat{\mathbf{r}}, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\, c \\ \widehat{\mathbf{r}} & = \langle\{\top_{Char}\}, \{\top_{Bool}\}, \{\top_{Int}\}\rangle \end{cases}$$

`Read-integer` works like `read-char`, but reads an integer from a port:

$$\frac{prim_l = \texttt{read-integer}}{\widehat{\varsigma} = ((prim_l\ \langle c, a\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \widehat{\mathbf{r}}, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\, c \\ \widehat{\mathbf{r}} & = \langle\{\top_{Int}\}, \{\top_{Bool}\}, \{\top_{Int}\}\rangle \end{cases}$$

Given an input port (a_1), an address (a_2), and a number of bytes (a_3) the primop `read-block` reads the specified number of bytes directly into the memory:

$$\frac{prim_l = \texttt{read-block}}{\widehat{\varsigma} = ((prim_l\ \langle c, a_1, a_2, a_3\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \widehat{\mathbf{r}}, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\, \widehat{\beta}\, \widehat{ve}\, \widehat{\sigma}\, \widehat{t'}\, c \\ \widehat{\mathbf{r}} & = \langle\{\top_{Char}\}, \{\top_{Bool}\}, \{\top_{Int}\}\rangle \end{cases}$$

Appendix D

Scheme Primops

The Scheme 48 system (version 1.7) has about 160 primitive operations implemented as instruction codes for the virtual machine. The CPS code derived from Scheme byte-code programs, however, only uses a subset of these primops. For example, primops that directly manipulate the stack, the memory representation of Scheme values, or environments do not occur in CPS programs. These primops are reserved for use by the standard byte-code compiler and the optimizer when translating a CPS program back to byte code.

Apart from these low-level primops there is a subset of primops that serve rather special purposes such as primops that access and manipulate the current continuation or access the low-level I/O system in the virtual machine. Typically, there are just a few library routines that employ these primops. The `current-cont` primop which returns the current continuation is an example: Only the implemenation of `call-with-current-continuation` needs to access the current continuation. Thus, there is exactly one library procedure that uses this primop. In these cases the analysis uses the syntactical description (see Section 6.3) to specify the semantics. Consequently, it is not necessary to specify a transition rule or a $\widehat{\mathcal{P}}_{Lang}$ clause for these primops.

For the implementation described in Chapter 6 I identified the relevant primops. The table below documents these primops. Most of the primops can occur both in trivial and in regular calls. For trivial calls, the following formula serves as a template:

$$\widehat{\mathcal{P}}_{Lang}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t}\ \mathcal{N}\langle a_0, \ldots, a_n\rangle = \{\widehat{\mathcal{R}}\}$$

Here, $\mathcal{N}$ is a primop name from the table below and $\widehat{\mathcal{R}}$ is the return value for this primop as listed in the table. If the primop occurs in a regular call, the following transition rule applies:

$$\frac{}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{c_i}\}, \langle\rangle, \langle\{\widehat{\mathcal{R}}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{c_i} & \in \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{\sigma}\ \widehat{t'}\ c \end{cases}$$

Name $\mathcal{N}$	n	**Description**
abstract boolean value: $\widehat{\mathcal{R}} = \top_{Bool}$		
char?	1	recognize character
complex?	1	recognize complex number
eof-object?	1	recognize EOF object
integer?	1	recognize integer number
number?	1	recognize number
rational?	1	recognize rational number
real?	1	recognize real number
exact?	1	check exactness of number value
char<?	2	compare characters
char=?	2	equality for characters
eq?	1	check for object identity
string=?	2	string equivalence
<	2	compare number values
<=	2	compare number values
=	2	equality for number values
location-defined?	1	check whether a certain binding is defined
stob-has-type?	2	check stored object for type
immutable?	1	check if stob field is mutable
unassigned-check	1	check whether binding ist initialized
abstract char value: $\widehat{\mathcal{R}} = \top_{Char}$		
string-ref	2	index string at given position
scalar-value->char	1	return char by its Unicode id
abstract string value: $\widehat{\mathcal{R}} = \widehat{string}$		
make-string	2	create new string
reverse-list->string	2	a common operation in the reader
exact abstract integer number $\widehat{\mathcal{R}} = (\texttt{exact}, \texttt{integer}, \top_{NumVal})$		
ceiling	1	real to integer conversion
truncate	1	real to integer conversion
floor	1	real to integer conversion
numerator	1	numerator of a rational number
denominator	1	denominator of a rational number
string-hash	1	compute hash value of a string
string-length	1	length of a string
bit-count	1	number of bits in an integer
byte-vector-length	1	length of a byte vector
arithmetic-shift	1	shift an integer n bits left or right
bitwise-and	2	combine integers bitwise with and
bitwise-ior	2	combine integers bitwise with inclusive or
bitwise-xor	2	combine integers bitwise with exclusive or
bitwise-not	1	bitwise not of an integer
byte-vector-ref	2	access a field of a byte vector
stob-len	1	number of fields in a stored object

Name $\mathcal{N}$	n	Description
inexact abstract real number $\widehat{\mathcal{R}} = (\texttt{inexact}, \texttt{real}, \top_{NumVal})$		
`log`	2	compute logarithm
`exp`	1	compute exponent e^n
`sin`	1	compute sinus function
`cos`	1	compute cosinus function
`tan`	1	compute tangent function
`asin`	1	compute asinus function
`acos`	1	compute acosinus function
`atan`	1	compute atangent function
inexact abstract real number $\widehat{\mathcal{R}} = \widehat{unspecific}$		
`collect`	0	force garbage collection
`string-set!`	3	destructively update character in string
`add-finalizer!`	2	connect value with finalization procedure
`byte-vector-set!`	3	destructively update byte in byte vector
`make-immutable!`	1	make binding immutable
inexact abstract real number $\widehat{\mathcal{R}} = \widehat{undef}$		
`unassigned`	0	return special *undefined* value

D.1 Arithmetic Primops

This section lists the transition rules for the arithmetic primops of Scheme supported by the flow analysis. For a discussion of the representation of abstract numbers see Section 4.4.2. The arithmetic primops may appear in trivial calls and regular calls. To save space I elided the definition of these primops as clauses for $\widehat{\mathcal{P}}_{Lang}$. However, deriving these clauses from the transition rules listed below is straightforward.

The following arithmetic primops retain the exactness of the arguments. That is, if the arguments are exact, the result is also exact.

$$\frac{\mathcal{N} \in \{\texttt{+}, \texttt{*}, \texttt{-}, \texttt{expt}\}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_0, a_1\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{r} & = \{(nt, e, \top_{NumVal}) \mid nt \in cntypes(ntypes(\widehat{d}))\} \\ e & = \begin{cases} \texttt{exact} & exact?(\widehat{d}) \\ \texttt{inexact} & inexact?(\widehat{d}) \\ \top_{Exact} & \text{otherwise} \end{cases} \end{cases}$$

The division operation `/` has a special rule:

$$\frac{}{\widehat{\varsigma} = ((\texttt{/}\ \langle c, a_0, a_1\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\,c \\ \widehat{r} & = \{(\top_{NumType}, \top_{Exact}, \top_{NumVal})\} \end{cases}$$

The number-theoretic primops `quotient` and `remainder` retain the precision but always return integer:

$$\frac{\mathcal{N} \in \{\texttt{quotient}, \texttt{remainder}\}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_0, a_1\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightharpoonup} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{r} & = \{(\texttt{integer}, e, \top_{NumVal}) \\ e & = \begin{cases} \texttt{exact} & exact?(\widehat{d}) \\ \texttt{inexact} & inexact?(\widehat{d}) \\ \top_{Exact} & \text{otherwise} \end{cases} \end{cases}$$

D.2 Primops for compound values

Section 4.4.2 explains the abstractions used to model Scheme 48's compound value types. This section summarizes the specification for these primops that deal with stobs.

Creating stobs `make-stob` creates a new stob value. There are three cases to distinguish: The first transition rule creates a record stob, the second a vector stob, and the third cases applies for the remaining stob types (pairs, cells, etc):

$$\frac{t_{Int} = \texttt{record}}{\widehat{\varsigma} = ((\texttt{make-stob}\ \langle c, t_{Int}, rt_{Int},, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightharpoonup} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_i \\ \widehat{f} & = \bigsqcup_{1\le i\le n} [i \mapsto ((\tau, \widehat{t'}), i)] \\ \widehat{ref} & = ((\tau, \widehat{t'}), \texttt{stob}) \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup [((\tau, \widehat{t'}), i) \mapsto \widehat{v}_i] \sqcup [\widehat{ref} \mapsto \{(\langle\texttt{record}, rt_{Int}\rangle, \widehat{f})\}] \end{cases}$$

$$\frac{t_{Int} = \texttt{vector}}{\widehat{\varsigma} = ((\texttt{make-stob}\ \langle c, t_{Int}, s, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\rightharpoonup} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_i \\ \widehat{f} & = [0 \mapsto ((\tau, \widehat{t'}), \texttt{elements})] \\ \widehat{ref} & = ((\tau, \widehat{t'}), \texttt{stob}) \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup [((\tau, \widehat{t'}), \texttt{elements}) \mapsto \bigsqcup_{1\le i\le n} \widehat{v}_i] \sqcup [\widehat{ref} \mapsto \{(\langle\texttt{vector}, s\rangle, \widehat{f})\}] \end{cases}$$

$$\frac{t_{Int} \notin \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = ((\texttt{make-stob}\ \langle c, t_{Int}, a_1, \ldots, a_n\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\{\widehat{ref}\}\rangle, \widehat{ve}, \widehat{\sigma}', \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_i \\ \widehat{f} & = \bigsqcup_{1 \le i \le n} [i \mapsto ((\tau, \widehat{t}'), i)] \\ \widehat{ref} & = ((\tau, \widehat{t}'), \texttt{stob}) \\ \widehat{\sigma}' & = \widehat{\sigma} \sqcup [((\tau, \widehat{t}'), i) \mapsto \widehat{v}_i] \sqcup [\widehat{ref} \mapsto \{(\langle t_{Int}\rangle, \widehat{f})\}] \end{cases}$$

Accessing a stob field There are three different primops that return the value of a stob field. `Checked-record-ref` accesses a field of a record stob and checks the record type as well as the stob type.

$$\frac{\mathcal{N} = \texttt{checked-record-ref}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_1 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v} \ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}} \ \wedge\ (\langle\texttt{record}, t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{r} & = \bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau', \widehat{t''}), i_{Int})) \end{cases}$$

The implementation of `vector-ref` uses the primop `stob-indexed-ref` to access an element of the vector. As discussed in Section 4.4, the analysis is not able to track the exact value for integer values used as indices. Consequently, `stob-indexed-ref` ignores the index j_{Int} and retrieves the value stored under the selector `0` which collapses all concrete values into a single abstract value:

$$\frac{t_{Int} = \texttt{vector} \qquad \mathcal{N} = \texttt{stob-indexed-ref}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}, j_{Int}\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d}_i\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t}')\}}$$

$$\text{where} \begin{cases} \widehat{t}' & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d}_i & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,c \\ \widehat{v} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t}'\,a_1 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v} \ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}} \ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{r} & = \bigsqcup_{((\tau, \widehat{t''}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau, \widehat{t''}), 0)) \end{cases}$$

If the stob is neither a record nor or a vector the following rule applies:

$$\frac{t_{Int} \notin \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = ((\texttt{stob-ref}\ \langle c, a_1, t_{Int}, i_{Int}\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\widehat{r}\rangle, \widehat{ve}, \widehat{\sigma}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{r} & = \displaystyle\bigsqcup_{((\tau, \widehat{t''}), s) \in \widehat{ref}} \widehat{\sigma}(((\tau, \widehat{t''}), i_{Int})) \end{cases}$$

Setting a stob field There are three primops that set a field of a stob. `Checked-record-ref` deals with stob that represent records:

$$\frac{\mathcal{N} = \texttt{checked-record-set!}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}, a_2\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_2 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle \texttt{record}, t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \displaystyle\bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \left[((\tau', \widehat{t''}), i_{Int}) \mapsto \widehat{new}\right] \end{cases}$$

Mutating vectors works similar but uses `stob-indexed-set!` primop which has an additional index j_{Int}:

$$\frac{t_{Int} = \texttt{vector} \qquad \mathcal{N} = \texttt{stob-indexed-set!}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, a_1, t_{Int}, i_{Int}, j_{Int}, a_2\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_2 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \displaystyle\bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} \left[((\tau', \widehat{t''}), i_{Int}) \mapsto \widehat{new}\right] \end{cases}$$

The primop `stob-set!` deals with the remaining stob types:

$$\frac{t_{Int} \neq \{\texttt{vector}, \texttt{record}\}}{\widehat{\varsigma} = ((\texttt{stob-set!}\ \langle c, a_1, t_{Int}, i_{Int}, a_2\rangle)_\tau, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{(\{\widehat{d_i}\}, \langle\rangle, \langle\rangle, \widehat{ve}, \widehat{\sigma'}, \widehat{t'})\}}$$

$$\text{where} \begin{cases} \widehat{t'} & = \widehat{tick}(\widehat{t}, \widehat{\varsigma}) \\ \widehat{d_i} & \in \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ c \\ \widehat{v} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_1 \\ \widehat{new} & = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{\sigma}\,\widehat{t'}\ a_2 \\ \widehat{ref} & = \{\widehat{u} :\ \widehat{u} \in \widehat{v}\ \wedge\ \widehat{u} \in \widehat{\mathbf{Ref}}\ \wedge\ (\langle t_{Int}\rangle, \widehat{f}) \in \widehat{\sigma}(\widehat{u})\} \\ \widehat{\sigma'} & = \widehat{\sigma} \sqcup \bigsqcup_{((\tau', \widehat{t''}), s) \in \widehat{ref}} [((\tau', \widehat{t''}), i_{Int}) \mapsto \widehat{new}] \end{cases}$$

D.3 Procedures with rest list argument

Scheme procedures have a fixed number of mandatory arguments and an optional rest list argument. This section summarizes the specification of the additional transition rule for n-ary procedures which is discussed in Section 4.4.2.

$$\frac{\text{length}(\widehat{c}^* \S \widehat{d}^*) \geq n - 1}{\widehat{\varsigma} = (\{((\lambda^{n\text{P}}(p_1 \ldots p_n)\ call)_\tau, \widehat{\beta}, \widehat{t_b})\}, \widehat{c}^*, \widehat{d}^*, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} (call, \widehat{\beta'}, \widehat{ve'}, \widehat{\sigma'}, \widehat{t})}$$

$$\text{where} \begin{cases} (\widehat{r}, \widehat{\sigma'}) & = \widehat{listify\text{-}args}((\widehat{c}^* \S \widehat{d}^*), n, \widehat{\varsigma}) \\ \widehat{\beta'} & = \widehat{\beta}[p_i \mapsto \widehat{t}] \\ \widehat{ve'} & = \widehat{ve}\ \sqcup \bigsqcup_{1 \leq j < n} [(p_j, \widehat{t}) \mapsto (\widehat{c}^* \S \widehat{d}^*)_j]\ \sqcup\ [(p_n, \widehat{t}) \mapsto \widehat{r}] \end{cases}$$

The transition rule requires the auxiliary function $\widehat{listify\text{-}args}$. Given an argument vector with n elements, an index, and a state $\widehat{listify\text{-}args}$ augments the store with a list that contains the elements from k to n of the argument vector.

$$\widehat{listify\text{-}args} : \widehat{\mathbf{D}}^* \times \mathbb{N} \times \widehat{\mathbf{Apply}} \to \widehat{\mathbf{D}} \times \widehat{\mathbf{Store}}$$

$$\widehat{listify\text{-}args}(\langle v_0, \ldots, v_n\rangle, k, \widehat{\varsigma}) = (\{\widehat{stob_k}\}, \widehat{\sigma'})$$

where

$$\begin{aligned} \widehat{stob_{n+1}} &= \{\widehat{empty}\} \\ \widehat{stob_i} &= ((\tau(a_i), \widehat{t}), \texttt{stob}) \\ \widehat{car_i} &= ((\tau(a_i), \widehat{t}), 0) \\ \widehat{cdr_i} &= ((\tau(a_i), \widehat{t}), 1) \\ \widehat{\sigma'} &= \widehat{\sigma}\ \sqcup \bigsqcup_{k \leq i \leq n} \left[\widehat{stob_i} \mapsto \{(\langle\texttt{pair}\rangle, [0 \mapsto \widehat{car_i}, 1 \mapsto \widehat{cdr_i}])\}\right] \\ &\quad \sqcup \bigsqcup_{k \leq i \leq n} \left[\widehat{car_i} \mapsto \widehat{v_i}\right] \\ &\quad \sqcup \bigsqcup_{k \leq i \leq n} \left[\widehat{cdr_i} \mapsto \widehat{stob_{i+1}}\right] \end{aligned}$$

with

$$\left((prim\ \langle a_0, \ldots, a_n\rangle), \widehat{\beta''}, \widehat{ve''}, \widehat{\sigma''}, \widehat{t''}\right) \widehat{\Rightarrow} \widehat{\varsigma}$$

D.4 Multiple return values

Calls to functions with multiple return values directly translate to calls using the `unknown-tail-call-with-values` primop and have the following form:

```
(unknown-tail-call-with-values cont proc receiver)
```

Here, `proc` is a procedure without arguments that returns n values and `receiver` is a procedure with n arguments called with the return values from `proc`. That is, this primop performs two procedure calls: A call to `proc` and subsequently a call to `receiver` — a behavior not compatible with the idea of CPS. To the flow analysis `unknown-tail-call-with-values` poses a problem since the analysis relies on CPS to assign a name to all intermediate values. The transition rule for `unknown-tail-call-with-values` (abbreviated as `utc-with-value`) must reestablish this condition first. The transition rule reads as follows:

$$\frac{\mathcal{N} = \texttt{utc-with-values} \quad p = (\lambda^{\mathsf{P}}()\ \ call)_{\tau'} \quad r = (\lambda^{\mathsf{P}}(a_1, \ldots, a_n)\ \ call)_{\tau''}}{\widehat{\varsigma} = ((\mathcal{N}\ \langle c, p, r\rangle)_{\tau}, \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t}) \widehat{\Rightarrow} \{((\texttt{let}\ \langle c''\rangle), \widehat{\beta}, \widehat{ve}, \widehat{\sigma}, \widehat{t})\}}$$

$$\text{where} \begin{cases} c' & = (\lambda^{\mathsf{C}}(b_1 \ldots b_n)\ (\texttt{unknown-call}\ \langle c, r, b_1, \ldots, b_n\rangle)_{\tau_1})_{\tau_2} \\ c'' & = (\lambda^{\mathsf{C}}()\ (\texttt{unknown-call}\ \langle c', p\rangle)_{\tau_3})_{\tau_4} \\ \tau_i & = \text{fresh labels} \\ b_i & = \text{fresh variables} \end{cases}$$

This rule is an *eval*-to-*eval* transition that translates the `unknown-tail-call-with-values` into CPS by connecting the value producing lambda p and value receiving lambda r with a new continuation lambda c'.

Bibliography

Agesen, O., The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism, in *9th European Conference on Object-Oriented Programming*, edited by W. G. Olthoff, no. 952 in Lecture Notes in Computer Science, pp. 2–26, Springer-Verlag, Åarhus, Denmark, 1995. 75

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers. Principles, Techniques, and Tools*, Addison-Wesley, 2007. 35

Ashley, J. M., The Effectiveness of Flow Analysis for Inlining, in *Proceedings International Conference on Functional Programming 1997*, edited by M. Tofte, pp. 99–111, ACM Press, New York, Amsterdam, The Netherlands, 1997. 10, 35, 36, 135

Bacon, D. F., S. L. Graham, and O. J. Sharp, Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, *26*, 345–420, 1994. 35

Burris, S., and H. Sankappanavar, *A Course in Universal Algebra*, millennium edition ed., 1999, revised edition available at http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html. Original printing: Graduate Texts in Mathematics. Springer Verlag, 1981. 41

Cejtin, H., S. Jagannathan, and S. Weeks, Flow-Directed Closure Conversion for Types Languages, pp. 56—71, 2000. 10

Consel, C., J. Lawall, and R. Marlet, Tempo Specializer — Users' manual, http://phoenix.labri.fr/software/tempo/doc/tempo-doc-user.html, 1998. 115

Consel, C., J. Lawall, and A.-F. Le Meur, A Tour of Tempo: A Program Specializer for the C Language, *Science of Computer Programming*, 2004. 115

Cousot, P., and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in *Proceedings of the 1997 ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 238–252, ACM Press, 1977. 36

Dubé, D., Demand-Driven Type Analysis for Dynamically-Typed Functional Languages, Ph.D. thesis, Université de Montréal, 2002. 134

Dybvig, R. K., The Development of Chez Scheme, in *Lawall* [2006], pp. 1–12. 10

Feeley, M., *Gambit-C, version 3.0, A portable implementation of Scheme*, 3.0 ed., 1998, http://www.iro.umontreal.ca/~gambit/doc/gambit-c.html. 127

Findler, R. B., and M. Felleisen, Contracts for Higher-Order Functions, in *Proceedings International Conference on Functional Programming 2002*, edited by S. Peyton-Jones, pp. 48–59, ACM Press, New York, Pittsburgh, PA, USA, 2002. 115

Flanagan, C., Effective Static Debugging via Componential Set-Based Analysis, Ph.D. thesis, Rice University, 1997. 114

Flanagan, C., and M. Felleisen, Componential set-based analysis, *ACM Transactions on Programming Languages and Systems*, *21*, 370–416, 1999. 114, 134

Frese, D., Multiprozessorunterstützung für Scheme 48, Master's thesis, Universität Tübingen, 2006. 135

Gabriel, R. P., *Performance and Evaluation of Lisp Systems*, MIT Press, Cambridge, MA, 1985. 127

Gasbichler, M., and M. Sperber, A Tractable Native-Code Scheme System, 2007, unpublished manuscript. 13, 17, 31

Ghuloum, A., and R. K. Dybvig, Generation-Friendly Eq Hash Tables, in *Proceedings of the 2007 Scheme and Functional Programming Workshop*, edited by D. Dubé, Freiburg, Germany, 2007. 113

Himsolt, M., GML: A Portable Graph File Format, http://www.infosun.fim.uni-passau.de/Graphlet/GML/gml-tr.html, 1995. 84

Jagannathan, S., and A. Wright, Effective Flow Analysis for Avoiding Runtime Checks, pp. 207–224, 1995. 35, 135

Jagannathan, S., and A. Wright, Flow-directed Inlining, in *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 193–205, ACM Press, Philadelphia, PA, USA, 1996. 10, 135

Jagannathan, S., and A. Wright, Polymorphic Splitting: An Effective Polyvariant Flow Analysis, *ACM Transactions on Programming Languages and Systems*, *20*, 166–207, 1998. 75

Jones, R., and R. Lins, *Garbage Collection*, John Wiley & Sons, 1996. 123

Kelsey, R., SRFI 9: Defining Record Types, http://srfi.schemers.org/srfi-9/, 1999. 68

Kelsey, R., W. Clinger, and J. Rees, Revised^5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, *11*, 7–105, 1998. 60, 61, 62

Kelsey, R. A., Compilation By Program Transformation, Ph.D. thesis, Yale University, 1989. 13, 17

Kelsey, R. A., Pre-Scheme: A Scheme Dialect for Systems Programming, *Tech. rep.*, NEC Research Institute, Inc., 1997. 13, 17, 53

Kelsey, R. A., and J. A. Rees, A Tractable Scheme Implementation, *Lisp and Symbolic Computation*, *7*, 315–335, 1995. 11, 13

Kennedy, A., Compiling with Continuations, Continued, in *Ramsey* [2007], pp. 177–190. 106

Lawall, J. (Ed.), *International Conference on Functional Programming*, ACM Press, New York, Portland, Oregon, USA, 2006. 163, 165

Matthews, J., R. Findler, M. Flatt, and M. Felleisen, A Visual Environment for Developing Context-Sensitive Term Rewriting Systems, in *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, edited by V. van Oostrom, vol. 3091 of *Lecture Notes in Computer Sciene*, Springer, 2004. 12, 83

Meunier, P., R. B. Findler, and M. Felleisen, Modular set-based analysis from contracts, pp. 218–231, 2006. 115

Might, M., and O. Shivers, Improving flow analyses via ΓCFA: Abstract garbage collection and counting, in *Lawall* [2006], pp. 13–25. 12, 24, 116, 117, 121

Might, M., and O. Shivers, Analyzing environment structure of higher-order languages using frame strings, *Theoretical Computer Science*, *375*, 137–168, 2007. 19, 24, 135

Mitchell, J., *Foundations for Programming Languages*, MIT Press, 1996. 41

Muchnick, S. S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, California, 1997. 35

Nielson, F., H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, 2nd corrected printing ed., Springer Verlag, 2005. 134, 137

Ramsey, N. (Ed.), *International Conference on Functional Programming*, ACM Press, New York, Freiburg, Germany, 2007. 165, 166

Serrano, M., Control Flow Analysis: a Functional Languages Compilation Paradigm, in *10th Acm Symposium on Applied Computing (SAC)*, pp. 118–122, Nashville, Tennessee, USA, 1995. 10

Serrano, M., *Bigloo—A "practical Scheme compiler"—User manual for version 2.6d*, 2004, `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html/`. 10

Shivers, O., Control-Flow Analysis of Higher-Order Languages or Taming Lambda, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1991, also technical report CMU-CS-91-145. 25, 36, 74, 124, 135

Sperber, M., W. Clinger, R. K. Dybvig, M. Flatt, and A. van Straaten, Revised[6] Report on the Algorithmic Language Scheme, `http://www.r6rs.org/final/r6rs.pdf`, 2007. 32

Stone, J. D., SRFI 8: `receive`: Binding to multiple values, http://srfi.schemers.org/srfi-8/, 1999. 60

van Horn, D., and H. G. Mairson, Relating Complexity and Precision in Control Flow Analysis, in *Ramsey* [2007], pp. 85–96. 75

van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science—Formal Models and Semantics*, vol. B, Elsevier Science Publishers, 1990. 137

Waddell, O., and R. K. Dybvig, Fast and Effective Procedure Inlining, in *Proceedings International Static Analysis Symposium, SAS'97*, edited by P. V. Hentenryck, no. 1302 in Lecture Notes in Computer Science, pp. 35–??, Springer-Verlag, Berkeley, California, 1997. 10

Waddell, O., D. Sarkar, and R. K. Dybvig, Fixing Letrec: A Faithful Yet Efficient Implementation of Scheme's Recursive Binding Construct, *Higher Order Symbolic Computation*, *18*, 299–326, 2005. 32

Wiese, R., M. Eiglsperger, and M. Kaufmann, yFiles: Visualization and Automatic Layout of Graphs, in *Graph Drawing*, pp. 453–454, 2001. 104, 132

Zeller, A., *Why programs fail. A guide to systematic debugging*, 1st ed., Morgan Kaufmann Publishers, 2005. 131