A Semantics of Python in Isabelle/HOL

James F. Ranson,
Howard J. Hamilton
and Philip W. L. Fong

Technical Report CS-2008-04
December 2008

Department of Computer Science
University of Regina
Regina, Saskatchewan, S4S 0A2
Canada

ISBN 978-0-7731-0657-4 (print)
ISBN 978-0-7731-0658-1 (online)

Abstract

As computers are deployed in increasingly diverse, numerous, and critical roles, the need for confidence in their hardware and software becomes more acute. Often, however, computer technologies, such as programming languages, lack a sufficiently formal definition to allow rigorous mathematical analysis of their properties. Even in cases where a formal definition is available, the theorems to be proven and the definition itself tend to have many cases and many details that are easily overlooked when writing a proof by hand. This has created interest in the mechanization of the proof process through the use of automated proof assistants.

In this report, we develop a formal definition for a programming language called IntegerPython, which is a subset of the Python language that supports integers, booleans, global variables, loops, modules, and nested functions. The definition takes the form of an operational semantics on a CEKS machine, which we embed in the Isabelle/HOL mechanized logic. We then prove an invariant of the CEKS machine in Isabelle/HOL. The report concludes with strategies for the efficient, executable implementation of the IntegerPython semantics and its extension into a semantics of the entire Python language.

Contents

Chapter 1

Introduction

In this chapter, we motivate and outline the research presented in this report. The first section motivates the use of mechanization in programming language metatheory in general and the study of the Python language in particular. The second section describes the nature and scope of the research presented. The third section outlines the remaining chapters.

1.1 Motivation

As computers are deployed in increasingly diverse, numerous, and critical roles in society, the need for confidence in their hardware and software becomes more acute. This has created interest in the application of rigorous mathematical analysis to computer technologies. Often, however, these technologies lack a sufficiently formal definition to allow for such analysis. For example, a programming language may be defined by informal documentation and a reference implementation of a compiler or interpreter. It can be difficult to reason about the behaviour of programs written in such a language, or about the effects of adding features to the language or changing how existing features are implemented. Even in cases where a formal definition is available, the definition itself and the results to be proven tend to be very complicated. Writing or verifying such a proof by hand can be daunting, and for the sake of brevity, proofs are often omitted from, or only summarized in published papers.

Fortunately, computers can be employed to assist with the composition and verification of proofs. A number of software packages are available for this purpose; they range from automatic theorem provers, which attempt to construct proofs with little or no human intervention, to proof assistants, which perform proofs by interactive or scripted execution of commands specified by a user. In either case, since the proof can be mechanically verified, we may be confident that it contains no missing cases, unsound inferences or overlooked details. Thus, even if a proof is omitted from a published paper, the mechanized proof script can be offered as an "electronic appendix" to improve confidence in the result, as suggested in [4]. This is not yet common practice in the Computer Science community, but efforts such as the POPLMARK challenge [4] indicate that opinion is swinging in favour of mechanized proof.

An advantage of mechanization is that it allows one to proceed more confidently with the

common tasks of programming language theory. For example, if one develops a transformation that takes programs from one language into another, and if suitable formal definitions exist for the source language and the target language, it may be possible to prove that the translation preserves the meaning of the programs. Moreover, since compilation can be viewed as translating programs from a readable source language into a target machine language, this technique could be used to prove the soundness of a compiler [5, 13, 26]. In the case of an interpreted language, one may propose a new interpreter that improves upon the reference interpreter in some way, such as in the time or space needed to evaluate a particular construct [9]; a mechanized model could be used to prove that the new interpreter provides the intended benefit while still giving the same results as the reference interpreter.

Even with mechanization, however, the task of constructing definitions for production quality, imperative programming languages is nontrivial. The logical feasibility of it is established by such works as [10, 29], and the thorough treatment of the Java™ language (http://java.sun.com/) in [25] demonstrates that it can indeed be done, but each language, or family of languages, presents its own notational challenges. In particular, scripting languages such as Python (http://www.python.org/) and Ruby (http://www.ruby-lang.org/) present different challenges from compiled languages such as Java and C. The former lack substantial treatment in the literature, and it is this gap that we expect to address in this report.

1.2 Nature and Scope of the Research

The goal of this report is to demonstrate a formal definition of a subset of a real-world scripting language and the use of such a definition in mechanically verifiable proofs. We accomplish this for the Python language by dividing its features into a four-stage development plan and completing the first stage. We also provide a deep embedding of the resulting definition in the Isabelle proof assistant and use it to prove a property of the chosen subset of Python. The development plan, with stages for IntegerPython, ObjectPython, StandardPython and ClassicPython, is outlined in Table 1.1, although the justification for this particular division of features is deferred to Chapter 5.

The definition of IntegerPython takes the form of an *operational semantics*, which is a formal description of the effect of each construct of a programming language on the state of an abstract machine. For this purpose, we employ a CEKS machine [8, 9, 10, 22], and our semantics accounts for integer and boolean values, Python's `None` value, **if** statements (conditionals), **while** statements (loops), functions and lambda expressions (collectively known as closures) and modules. We base the semantics on the Python Reference Manual [33], and where this manual is ambiguous, we clarify it by refering to sample Python programs and their output when run on version 2.5 of the reference Python interpreter, which is found at http://www.python.org/.

1.3 Outline

The remainder of this report proceeds as follows: In Chapter 2, we introduce notation and describe related work. The first section introduces the CEKS machine; the meaning of each of the four

Table 1.1: Proposed stages of feature compliance

Stage I *IntegerPython*

- integers, booleans and `None`
- loops, conditionals, closures and modules

Stage II *ObjectPython*

- partial type hierarchy
- strings and dictionaries
- "new-style" classes
- modules and closures as general objects
- exceptions

Stage III *StandardPython*

- complete "standard type hierarchy", less "classic" objects
- all source constructs not already covered

Stage IV *ClassicPython*

- the "classic" object model (pre Python 2.2)

registers is explained and an operational semantics for a small toy language is developed. The second section discusses automated theorem provers in general and compares two in particular, ACL2 and Isabelle, in terms of their use by other authors and their suitability for this research.

In Chapter 3, we develop an operational semantics for IntegerPython. The first section discusses the general structure of IntegerPython programs and presents some concrete examples. The second section gives the complete abstract syntax of IntegerPython and an informal description of the meaning of each construct. The third section describes the notation for the IntegerPython CEKS machine. The fourth section completes the operational semantics by describing the transition rules of the CEKS machine.

In Chapter 4, the use of the IntegerPython semantics in mechanized proof scripts is discussed. The first section describes an embedding of the semantics in the Isabelle/HOL mechanized logic. The second section discusses techniques for program verification. The third section proves an invariant of the CEKS machine.

Finally, in Chapter 5, we present conclusions and give suggestions for future work. The first section analyzes the efficiency of the IntegerPython semantics and gives strategies for the implementation of an efficient interpreter. The second section explains the division of features in the four-stage development plan and suggests how the remaining features could be implemented. The third section concludes the report with a summary of the key points from each chapter.

Chapter 2

Background and Related Work

In this chapter, we provide background on the CEKS abstract machine and the use of mechanized proving technology in programming language theory. The first section introduces the CEKS machine, defines a toy language and develops an operational semantics for it on a CEKS machine. The second section discusses automated theorem provers and proof assistants.

2.1 The CEKS Machine

The CEKS machine [8, 9, 10, 22] is an abstract machine with four *registers*, named C, E, K and S. The notation for each register varies between authors and depends on the source language that is being run on the machine, but the purpose of each register is generally the same and can be described as follows:

C is the *control register*, and it contains an *instruction*, a *semantic value* (or simply, *value*) or an *error condition*. In general, the operation of the machine *reduces* instructions to smaller instructions and eventually to semantic values. Putting an instruction into C is known as *scheduling* the instruction.

E is the *environment register*. Mathematically, an *environment* is a finite partial function from the set of variable names to a countably infinite set of *locations*. Names for which the environment is defined are said to be *bound* by the environment, and the term *binding* refers either to the name-location pair or to the location alone, depending on the context. Semantically, an environment serves as a record of which variable names may be expected to have a value at a particular program point.

K is the *continuation register*. A *continuation* is a data structure that represents a deferred computation step; generally, it consists of a name and a number of parameters, which represent the data that must be *remembered* in order to complete the deferred computation. The K register is typically treated as a stack of continuations; as instructions are reduced, continuations may be pushed onto the stack, and once C contains a value, a continuation is popped and *applied* to the value to determine the next machine state.

S is the *store register*. A *store* is a finite partial function from the set of locations to the set of values, and so it works in conjunction with an environment to map variable names to values. This indirection allows the machine to represent *aliases*, which are names in different environments, or distinct names in the same environment, that refer to the same location.

States of the machine are denoted by a quadruple, $\langle C, E, K, S\rangle$, and programs are executed by transitioning between states according to a set of machine state pairs called the *transition relation*. In general, the transition relation is infinite but induced by a finite set of transition rules. A *transition rule* is an abstract (i.e. containing *metavariables*) machine state pair, so that members of the transition relation are given by instantiating the metavariables of the transition rule. When a context-free grammar is used to specify the CEKS machine, metavariables are named for the non-terminal symbols of the grammar, possibly with subscripts for distinction; they may assume any value that can be derived from the associated non-terminal symbol. We say that a transition rule *applies* to a particular machine state if the metavariables of the rule's first element can assume values that make it equal to the state. Furthermore, transition rules are categorized by the states to which they apply: if a transition rule applies to states where C contains an instruction (value), it is called a *reduction rule* (respectively, *continuation rule*).

Recall that an operational semantics for a programming language is a description of the effect of each language construct on the state of an abstract machine. Therefore, if a CEKS machine is used, an operational semantics consists of the following:

- Notations for C, E, K and S.
- A finite set of transition rules.
- A description of starting states and terminal states.

To illustrate the process of defining an operational semantics, consider the toy language in Figure 2.1. It features named variables ($\langle \mathbf{Var} : \nu\rangle$), integer constants ($\langle \mathbf{Const} : i\rangle$), arithmetic operations ($\langle \mathbf{Bop} : b, e, e\rangle$), assignment ($\langle \mathbf{Assign} : \nu, e\rangle$), conditional execution ($\langle \mathbf{IfPos} : e, s, s\rangle$, which executes the first statement if e reduces to a positive integer, and the second if e reduces to zero or a negative integer), printing ($\langle \mathbf{Print} : e\rangle$), sequential execution ($\langle \mathbf{Statements} : s, s\rangle$) and iteration ($\langle \mathbf{WhilePos} : e, s\rangle$, which runs s repeatedly as long as e reduces to a positive integer). The description in Figure 2.1 is known as abstract syntax. *Abstract syntax* is a description of how programs may be constructed, in terms of fundamental building blocks that represent the essential paradigms of the language. *Concrete syntax*, on the other hand, specifies how programs may be constructed in terms of sequences of characters in a text file. In defining the semantics of the toy language, we omit the concrete syntax and work directly from the abstract syntax.

Proceeding with the definition of the toy CEKS machine, the control register may contain an instruction (i.e. an expression or statement), a semantic value or an error condition. Thus, the domain of the control register is defined by the following grammar:

$$\text{TOYCONTROL} \quad \ni \quad C \quad ::= \quad i \mid s \mid e \mid \langle \mathbf{Error} : C\rangle.$$

$$
\begin{array}{rcccl}
 & & i & \in & \textsc{Integer} \\
 & & \nu & \in & \textsc{ToyIdentifier} \\
 & & & & \\
\textsc{ToyExpression} & \ni & e & ::= & \langle \mathbf{Var} : \nu \rangle \\
 & & & | & \langle \mathbf{Const} : i \rangle \\
 & & & | & \langle \mathbf{Bop} : b, e, e \rangle \\
\textsc{ToyBinaryOperation} & \ni & b & ::= & \texttt{Add} \\
 & & & | & \texttt{Sub} \\
 & & & | & \texttt{Mul} \\
 & & & | & \texttt{Div} \\
 & & & | & \texttt{Mod} \\
\textsc{ToyStatement} & \ni & \mathrm{s} & ::= & \langle \mathbf{Assign} : \nu, e \rangle \\
 & & & | & \langle \mathbf{IfPos} : e, s, s \rangle \\
 & & & | & \langle \mathbf{Print} : e \rangle \\
 & & & | & \langle \mathbf{Skip} \rangle \\
 & & & | & \langle \mathbf{Statements} : s, s \rangle \\
 & & & | & \langle \mathbf{WhilePos} : e, s \rangle
\end{array}
$$

Figure 2.1: A toy language for integer arithmetic

Partial functions (i.e. for environments and stores) are represented inductively. That is, a partial function is either an empty function (i.e. it does not provide a mapping for any input), or it is an extension of another partial function. An *extension* of a partial function provides the same mapping (or lack of mapping) as the function itself for every input except one. Environments and stores are defined as partial functions on their respective domains as follows:

$$l \quad \in \quad \textsc{Location}$$

$$E \quad ::= \quad \diamond \mid E[\nu \mapsto l]$$

$$S \quad ::= \quad \circ \mid S[l \mapsto i].$$

Here LOCATION can be any countably infinite set, such as $\mathbb{N}$ itself, or perhaps $\{\langle \mathbf{Location} : n \rangle \mid n \in \mathbb{N}\}$. The symbol $\diamond$ denotes an empty environment, and $E[\nu \mapsto l]$ denotes an extension of the environment E that binds the name ν to the location l. The application of an environment to a name is defined inductively as follows:

$$E[\nu' \mapsto l](\nu) = \begin{cases} l & \text{if } \nu = \nu' \\ E(\nu) & \text{if } \nu \neq \nu'. \end{cases}$$

The set of names for which a given environment contains a mapping, denoted $\mathit{dom}(E)$, is defined by the following equations:

$$\begin{array}{rcl} \mathit{dom}(E[\nu \mapsto l]) & = & \{\nu\} \cup \mathit{dom}(E) \\ \mathit{dom}(\diamond) & = & \emptyset. \end{array}$$

A store is applied to a location as follows:

$$S[l' \mapsto i](l) \quad = \quad \begin{cases} i & \text{if } l = l' \\ S(l) & \text{if } l \neq l'. \end{cases}$$

Finally, the domain of a store is defined by the following equations:

$$\begin{aligned} \mathit{dom}(S[l \mapsto i]) &= \{l\} \cup \mathit{dom}(S) \\ \mathit{dom}(\circ) &= \emptyset. \end{aligned}$$

The following are the continuations of the machine:

$$\begin{aligned} K \quad ::= \quad & \langle \mathsf{BopLeft} : b, e, K \rangle \\ | \quad & \langle \mathsf{BopRight} : i, b, K \rangle \\ | \quad & \langle \mathsf{Assign} : \nu, K \rangle \\ | \quad & \langle \mathsf{If} : s, s, K \rangle \\ | \quad & \langle \mathsf{Print} : K \rangle \\ | \quad & \langle \mathsf{Statement} : s, K \rangle \\ | \quad & \langle \mathsf{WhileTest} : e, s, K \rangle \\ | \quad & \langle \mathsf{WhileRun} : e, s, K \rangle \\ | \quad & \langle \mathsf{Halt} \rangle. \end{aligned}$$

Each one, with the exception of $\langle \mathsf{Halt} \rangle$, contains another continuation, so that continuations can be stacked by nesting them. In general, the grammar of continuations is derived by considering the steps involved in reducing each kind of instruction to a value. The $\langle \mathbf{Var} : \nu \rangle$ expression is reduced in one step to $S(E(\nu))$, and $\langle \mathbf{Const} : i \rangle$ to i, so no continuation is needed. However, binary operations require information to be saved in a continuation, because both the left and right operands are expressions and C can hold only one at a time. During the reduction of the left operand expression, the operator and the right operand expression are remembered in the $\langle \mathsf{BopLeft} : b, e, K \rangle$ continuation. Likewise, during the reduction of the right operand expression, the operator and the left operand value are remembered in the $\langle \mathsf{BopRight} : i, b, K \rangle$ continuation. Assignment requires the reduction of the right-hand side to take place before the assignment can occur, so the left-hand side (i.e. the variable name) is remembered in the $\langle \mathsf{Assign} : \nu, K \rangle$ continuation. Applying $\langle \mathsf{If} : s, s, K \rangle$ to a positive value causes the first statement to be executed, and a non-positive value, the second. The $\langle \mathsf{Print} : K \rangle$ continuation serves only to indicate that a printing operation is in progress, since the value of a printed expression is not actually used by the machine. The $\langle \mathsf{Statement} : s, K \rangle$ continuation arises from the fact that statements come in pairs in $\langle \mathbf{Statements} : s, s \rangle$; the second is saved in a continuation while the first is reduced. A while loop consists of two stages: evaluating the test expression and running the loop body. When $\langle \mathsf{WhileTest} : e, s, K \rangle$ is applied to a value, the value is taken as the result of the test expression, and the resulting machine state depends on whether the value is positive. In the case of $\langle \mathsf{WhileRun} : e, s, K \rangle$, the test expression is scheduled and $\langle \mathsf{WhileRun} : e, s, K \rangle$ is replaced with $\langle \mathsf{WhileTest} : e, s, K \rangle$, regardless of the exact value in C.

Lastly, the CEKS machine states are defined by the following grammar:

$$R_{var}: \quad \langle\langle \mathbf{Var}: \nu\rangle, E, K, S\rangle \longrightarrow_{toy} \begin{cases} \langle S(E(\nu)), E, K, S\rangle & \text{if } \nu \in \mathit{dom}(E) \text{ and } E(\nu) \in \mathit{dom}(S) \\ \langle\langle \mathbf{Error}: \langle \mathbf{Var}: \nu\rangle\rangle, E, K, S\rangle & \text{if } \nu \notin \mathit{dom}(E) \text{ or } E(\nu) \notin \mathit{dom}(S) \end{cases}$$

$$\begin{aligned}
R_{const}: &\quad \langle\langle \mathbf{Const}: i\rangle, E, K, S\rangle \longrightarrow_{toy} \langle i, E, K, S\rangle \\
R_{bop}: &\quad \langle\langle \mathbf{Bop}: b, e_l, e_r\rangle, E, K, S\rangle \longrightarrow_{toy} \langle e_l, E, \langle \mathsf{BopLeft}: b, e_r, K\rangle, S\rangle \\
R_{assign}: &\quad \langle\langle \mathbf{Assign}: \nu, e\rangle, E, K, S\rangle \longrightarrow_{toy} \langle e, E, \langle \mathsf{Assign}: \nu, K\rangle, S\rangle \\
R_{if}: &\quad \langle\langle \mathbf{IfPos}: e, s_t, s_f\rangle, E, K, S\rangle \longrightarrow_{toy} \langle e, E, \langle \mathsf{If}: s_t, s_f, K\rangle, S\rangle \\
R_{print}: &\quad \langle\langle \mathbf{Print}: e\rangle, E, K, S\rangle \longrightarrow_{toy} \langle e, E, \langle \mathsf{Print}: K\rangle, S\rangle \\
R_{skip}: &\quad \langle\langle \mathbf{Skip}\rangle, E, K, S\rangle \longrightarrow_{toy} \langle 0, E, K, S\rangle \\
R_{statements}: &\quad \langle\langle \mathbf{Statements}: s_1, s_2\rangle, E, K, S\rangle \longrightarrow_{toy} \langle s_1, E, \langle \mathsf{Statement}: s_2, K\rangle, S\rangle \\
R_{while}: &\quad \langle\langle \mathbf{WhilePos}: e, s\rangle, E, K, S\rangle \longrightarrow_{toy} \langle e, E, \langle \mathsf{WhileTest}: e, s, K\rangle, S\rangle
\end{aligned}$$

Figure 2.2: Reduction rules for the toy CEKS machine

$$\textsc{ToyState} \quad \ni \quad M \quad ::= \quad \langle C, E, K, S\rangle$$

A suitable start state for the reduction of an instruction $c \in \textsc{ToyExpression} \cup \textsc{ToyStatement}$ is $\langle c, \diamond, \langle \mathsf{Halt}\rangle, \circ\rangle$; terminal states are derived from $\langle i, E, \langle \mathsf{Halt}\rangle, S\rangle$ and $\langle\langle \mathbf{Error}: C\rangle, E, K, S\rangle$.

The reduction rules are given in Figure 2.2 and may be interpreted as follows: R_{var} reduces a variable name to the associated value by looking up the name in the environment ($E(\nu)$) and looking up the corresponding location in the store ($S(E(\nu))$); if either lookup would fail, an error condition is put into the C register. R_{const} simply reduces an integer constant expression to an integer value. R_{bop} begins the reduction of a binary expression by scheduling the left operand expression and remembering the operator, right operand and previous value of K. R_{assign} schedules the right-hand side of the assignment and remembers the variable name. R_{if} schedules the test expression and remembers the possible statements. R_{print} schedules the printed expression and pushes the $\langle \mathsf{Print}: K\rangle$ continuation, so that the printed output of a program can be found by examining its trace for states of the form $\langle i, E, \langle \mathsf{Print}: K\rangle, S\rangle$. Since the $\langle \mathbf{Skip}\rangle$ statement does nothing, R_{skip} reduces it to 0. (In general, statements may reduce to any value, but here the convention is to reduce them to 0.) $R_{statements}$ executes a pair of statements by scheduling the first and remembering the second. Finally, R_{while} schedules the test expression of the loop and remembers it along with the loop body.

The continuation rules in Figure 2.3 describe the effect of applying each of the continuations that may exist in K to the value in C. $C_{bopleft}$ stores the value as the value of the left operand and schedules the right operand. $C_{bopright}$ takes the value of the right operand and performs the binary operation, where the function

$$B : \textsc{Integer} \times \textsc{ToyBinaryOperation} \times \textsc{Integer} \rightarrow \textsc{ToyControl}$$

$$C_{\textit{bopleft}}: \quad \langle i, E, \langle \textsf{BopLeft}: b, e, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \langle e, E, \langle \textsf{BopRight}: i, b, K\rangle, S\rangle$$
$$C_{\textit{bopright}}: \quad \langle i_r, E, \langle \textsf{BopRight}: i_l, b, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \langle B(i_l, b, i_r), E, K, S\rangle$$

$$C_{\textit{assign}}: \quad \langle i, E, \langle \textsf{Assign}: \nu, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \begin{cases} \langle 0, E[\nu \mapsto l], K, S[l \mapsto i]\rangle & \text{if } \nu \notin \textit{dom}(E), \text{ where } l \notin \textit{dom}(S). \\ \langle 0, E, K, S[E(\nu) \mapsto i]\rangle & \text{if } \nu \in \textit{dom}(E). \end{cases}$$

$$C_{\textit{if}}: \quad \langle i, E, \langle \textsf{If}: s_t, s_f, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \begin{cases} \langle s_t, E, K, S\rangle & \text{if } i > 0 \\ \langle s_f, E, K, S\rangle & \text{if } i \leq 0 \end{cases}$$

$$C_{\textit{print}}: \quad \langle i, E, \langle \textsf{Print}: K\rangle, S\rangle \longrightarrow_{\textit{toy}} \langle 0, E, K, S\rangle$$
$$C_{\textit{statement}}: \quad \langle i, E, \langle \textsf{Statement}: s, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \langle s, E, K, S\rangle$$
$$C_{\textit{whilerun}}: \quad \langle i, E, \langle \textsf{WhileRun}: e, s, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \langle e, E, \langle \textsf{WhileTest}: e, s, K\rangle, S\rangle$$

$$C_{\textit{whiletest}}: \quad \langle i, E, \langle \textsf{WhileTest}: e, s, K\rangle, S\rangle \longrightarrow_{\textit{toy}} \begin{cases} \langle s, E, \langle \textsf{WhileRun}: e, s, K\rangle, S\rangle & \text{if } i > 0 \\ \langle 0, E, K, S\rangle & \text{if } i \leq 0 \end{cases}$$

Figure 2.3: Continuation rules for the toy CEKS machine

defines the semantics of binary operations in the obvious way, with $B(i, \texttt{Div}, 0) = B(i, \texttt{Mod}, 0) = \langle \textbf{Error} : 0 \rangle$. C_{assign} has two cases, depending on whether the variable is already bound in the environment; if it is, only the store is updated with the new value of the variable; if it is not, the variable is bound to a fresh location $l \notin dom(S)$. C_{if} chooses the statement to execute based on the positivity of the value in C. C_{print} makes the `print` statement reduce to 0, just as the other statements do. $C_{statement}$ schedules the second of a pair of statements, after the first has been fully reduced. Likewise, $C_{whilerun}$ schedules the test expression of the loop after the body has been fully reduced. Finally, $C_{whiletest}$ determines if the loop body should be scheduled again or if the loop should terminate.

2.2 Automated Theorem Provers

Often, theorems about programming languages have many cases, each with many details that are easily overlooked when writing a proof by hand. Automating the proving process reduces the likelihood of human error, because the resulting proofs can be verified mechanically. In general, two types of software are available for automated theorem proving: automatic theorem provers and proof assistants. An *automatic theorem prover* is a program that attempts to prove theorems without human intervention or assistance, whereas a *proof assistant* provides mechanical checking of *proof scripts* written by a human. Some proof assistants also provide semi-automatic proof methods, so that the human involvement is limited to coaching when the automatic methods fail.

A number of automated theorem provers exist, but the following two were considered because of their previous use in the analysis of production-quality programming languages [7, 11, 15, 16, 19, 21, 23, 24, 25, 27]:

ACL2, or A Computational Logic for Applicative Common Lisp, is a product of Matt Kaufmann and J Strother Moore at the University of Texas at Austin; it is freely available at http://www.cs.utexas.edu/~moore/acl2/. ACL2 allows the user to state theorems about programs written in a purely functional (i.e. applicative) subset of Common Lisp [30], and to prove those theorems automatically. It provides a syntax for specifying hints, but otherwise it searches for proofs automatically, according to the algorithm described in [6], which includes heuristics for automatic inductive proof. Also, since the analysis of a programming language in ACL2 typically involves implementing an interpreter for it in Common Lisp, a by-product of the analysis is a stand-alone executable interpreter for the language.

Isabelle/HOL, a joint work of Larry Paulson at the University of Cambridge and Tobias Nipkow at the Technical University of Munich, is a generic proof assistant for higher order logic; it is freely available at http://isabelle.in.tum.de/. It features a readable syntax for proof scripts and a rich collection of proof methods, including the *auto* method, which provides semi-automatic theorem proving, although it does not attempt induction.

To determine the most suitable candidate for our work, we compared ACL2 and Isabelle/HOL on the following criteria:

- Expressiveness of the logic.

```
(defthm pascal-triangle-binomial
  (implies
    (and (integerp k) (integerp n) (<= 0 k))
    (equal
      (sumlist (binomial-expansion-pascal-triangle x y k n))
      (sumlist (binomial-expansion x y (1+ k) (1+ n)))))
  :hints (("Goal"
       :induct (binomial-expansion-pascal-triangle x y k n)
       :in-theory (disable choose expt
                           right-unicity-of-1-for-expt
                           expt-minus distributivity-of-expt-over-*
                           exponents-multiply
                           functional-commutativity-of-expt-/-base
                           exponents-add
                           exponents-add-for-nonneg-exponents))
      ("Subgoal *1/1''"
       :use ((:instance choose-reduction
                        (k (+ 1 k))
                        (n (+ 1 n)))))))
```

Figure 2.4: A proof in ACL2 [2]

- Readability and writability of proof scripts.
- Readability and writability of function definitions.
- Use by other authors.

With respect to expressiveness of the logic, the ACL2 logic is a "full first order logic," augmented with "encapsulation and functional instantiation, [which] provide a second order aspect [by allowing] quantification over functions" [12]. In contrast, Isabelle/HOL is based on simply typed lambda calculus as a formulation of higher order logic. Thus, it permits not only quantification over functions on individuals (i.e. second order logic), but also quantification over functions on functions, etc. A detailed discussion of higher order logic is found in [3], but in summary, Isabelle's logic is more expressive that ACL2's.

With respect to readability and writability of proof scripts, it is difficult to compare ACL2 and Isabelle/HOL, because ACL2's "proof scripts" are often just the statement of theorems for automatic proof. However, when the automatic methods fail, ACL2's syntax for specifying hints can be cryptic. The proof in Figure 2.4, which is an excerpt from a proof of the Binomial Theorem in ACL2's theory library [2], demonstrates some of the weaknesses of the syntax. Most notably, goals are addressed by number, and the reader of the proof may not know what "Subgoal *1/1''" actually refers to. In contrast, Isabelle/HOL's syntax encourages a structured, step-by-step development where intermediate results are explicitly stated and proven, and the proof can be understood by

theorem *binomial*: $(a+b::nat)\hat{}n = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
proof (*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
have *decomp*: $\{0..n+1\} = \{0\} \cup \{n+1\} \cup \{1..n\}$
by (*auto simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
have *decomp2*: $\{0..n\} = \{0\} \cup \{1..n\}$
by (*auto simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
have $(a+b::nat)\hat{}(n+1) = (a+b) * (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
using *Suc* **by** *simp*
also have $\ldots = a*(\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k)) +$
$b*(\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
by (*rule nat-distrib*)
also have $\ldots = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}(k+1) * b\hat{}(n-k)) +$
$(\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k+1))$
by (*simp add*: *setsum-right-distrib mult-ac*)
also have $\ldots = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n+1-k)) +$
$(\sum k{=}1..n+1.\ (n\ choose\ (k-1)) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*:*setsum-shift-bounds-cl-Suc-ivl Suc-diff-le*
del:*setsum-cl-ivl-Suc*)
also have $\ldots = a\hat{}(n+1) + b\hat{}(n+1) +$
$(\sum k{=}1..n.\ (n\ choose\ (k-1)) * a\hat{}k * b\hat{}(n+1-k)) +$
$(\sum k{=}1..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*: *decomp2*)
also have
$\ldots = a\hat{}(n+1) + b\hat{}(n+1) + (\sum k{=}1..n.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*: *nat-distrib setsum-addf binomial.simps*)
also have $\ldots = (\sum k{=}0..n+1.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
using *decomp* **by** *simp*
finally show *?case* **by** *simp*
qed

Figure 2.5: A proof in Isabelle/HOL [1]

Table 2.1: Comparison of ACL2 and Isabelle/HOL

Criterion	Result
Expressiveness	Isabelle/HOL is more expressive than ACL2.
Proof scripts	Isabelle/HOL is favoured for its document preparation and readable syntax.
Function definitons	Isabelle/HOL is favoured for its syntactic typing.
Use by other authors	ACL2 is favoured for operational analysis because of its executability.

reading it on paper. This is illustrated by the proof in Figure 2.5, which is an excerpt from the Isabelle/HOL proof of the Binomial Theorem [1]. Here, each step of the proof is explicitly stated, as it would be in a mathematical paper. The proof also demonstrates Isabelle/HOL's typesetting annotations and automatic document preparation, which provide syntax highlighting and the conversion of certain source constructs into mathematical symbols (e.g. `\<Sum>` in the ASCII source becomes Σ in the printed document).

With respect to readability and writabality of function definitions, Isabelle/HOL benefits from being syntactically typed. A function that is designed to take an integer as an argument, for example, need not consider that it may be passed a string instead. In contrast, Common Lisp is syntactically untyped, so an ACL2 function is often burdened with establishing the type of its parameters by an `if` or `cond` statement and providing a return value for improper input that syntactic typing could have ruled out.

With respect to use by other authors, both ACL2 and Isabelle/HOL are used for a variety of topics related to ours. Applications of ACL2 to operational analysis include [15, 16, 18, 19, 21], in which the Java Virtual Machine (JVM) specification [14] is partially implemented in ACL2, and Java programs are analyzed by compiling them into bytecode and executing them on this JVM implementation. Isabelle/HOL is also used in the operational analysis of Java [24, 25], although it does not (readily) yield executable models, as ACL2 does. Other topics to which ACL2 has been applied include type soundness [31] and hardware verification [20]. Isabelle/HOL has been applied to type soundness [23, 25], axiomatic program verification [25, 29] and compiler verification [5].

The summary of the comparison is presented in Table 2.1. Isabelle/HOL wins on three out of four points and therefore was used in the research reported here. We decided to sacrifice executability in favour of the other advantages of Isabelle/HOL.

Chapter 3

An Operational Semantics of IntegerPython

In this chapter, we develop an operational semantics of the IntegerPython language. The first section presents an overview of IntegerPython and describes its characteristic features. In the second section, the complete abstract syntax of IntegerPython is defined, along with an informal discussion of the semantics of each language construct. The third section defines a CEKS machine for IntegerPython, using the abstract syntax as an instruction representation. Finally, the fourth section gives the transition rules of this CEKS machine.

3.1 Overview of IntegerPython

In general, IntegerPython is intended to be a subset of Python. That is, every IntegerPython program should be a syntactically valid Python program, and it should have the same meaning in Python as it has in IntegerPython. The concrete syntax of IntegerPython, which is specified in Appendix A, can be derived from the concrete syntax of Python by removing some production rules and simplifying certain others. It follows that any concrete IntegerPython program is also a concrete Python program, at least as far as the grammar is concerned. Exceptions arising from Python's enforcement of additional constraints are noted and explained here and in the following section. That the meaning of programs is preserved from IntegerPython to Python is not as easy to prove, but we believe this to be true, except where noted, and we support the claim with examples and quotations from the Python Language Reference [33].

IntegerPython code is grouped into functions, modules and programs. A *program* is a collection of named modules, where one module is designated the *main module*. Informally, when a program consists of only one module, that module may be called a program. A *module* consists of a list of IntegerPython statements (the *module body*) that is at the top level lexically, in that it is not contained in a function or another module. (An implementation could put modules into seperate files in a filesystem.) Lexically, a *function* is a list of statements (the *function body*) grouped under a **def** statement, which also specifies a possibly empty list of variable names that are the *formal parameters* of the function. Function definitions occur within modules, and they may be nested within other function definitions. Execution of a program consists of the execution of the body of the main module, which may refer to other modules by a process known as importing, as described

below.

With respect to variables, IntegerPython does not require variable names to be declared before they are used. Variables are created when the first *binding statement* (e.g. assignment, function definition) for a *name* in a given namespace is executed. Mathematically, a *namespace* is a partial function from names to memory locations; lexically, a namespace corresponds to a function body (called a *local namespace*) or a module body (called a *global namespace*). A name is said to be *bound* in a local namespace if a binding statement for that name occurs anywhere within the namespace or it is a formal parameter of the corresponding function. In a global namespace, however, the binding of names is dynamic and depends on whether a binding statement has actually been executed. A name that occurs in a local namespace but is not bound there is called *free*, and free names are *resolved* in the innermost enclosing namespace where they are bound. Because the bound names of local (not global) namespaces can be statically determined (Appendix B.8), resolution from enclosing local namespaces occurs at the time of the function's definition, so that the result of function definition is a value called a *closure*, which contains the following elements:

- A function body.
- A list of formal parameters.
- A list of bound names.
- A reference to a new namespace in which the resolved names are bound.
- A reference to the global namespace in which the function was defined.

The list of bound names can be computed statically from the function body and is included in the closure only for the sake of efficiency. It is used at invocation-time to initialize the local namespace, so that the use of a bound name before its binding statement can be detected as an error, not as a reference to a global variable. During an execution of the function body, free names that have not been statically resolved are resolved on-demand from the global namespace.

The use of nested functions with free names is demonstrated by the program in Figure 3.1, where the `intervening` function is nested within the `outer` function, and the `inner` function is nested within `intervening`. (Note the use of indentation to delimit the function bodies.) Execution of this program proceeds as follows:

1. The `def outer():` statement defines a function and creates a new global variable, `outer`, to store the resulting closure. The body of the function is not executed at this time, so the nested functions (`intervening` and `inner`) remain undefined.
2. The `x = 9` statement assigns the value 9 to a new global variable `x`.
3. The `f = outer()` statement invokes `outer` and creates a global variable `f` to store its return value. The body of `outer` is executed as follows:
   (a) The `intervening` function is defined in the local namespace of `outer`. Because `x` is free in `inner`, and is not bound in `intervening`, it is considered free in

```
def outer():
    def intervening():
        def inner():
            print x
        return inner
    x = 5
    return intervening()

x = 9 # Global x
f = outer()
f()
```

Figure 3.1: An IntegerPython program with nested functions

```
import freegap
y = freegap.x
freegap.x = 10
print y, freegap.x
import freegap as freegap2
print freegap2.x
```

Figure 3.2: An IntegerPython module that imports another module

`intervening`. However, `x` is bound in `outer`, even though the assignment to `x` has not yet occurred. Therefore, the `x` in `intervening` is resolved to the `x` in `outer`.

(b) The local variable `x`, which already exists in an uninitialized state, is assigned the value 5.

(c) The return value of `intervening` is returned to the caller of `outer`. `intervening` defines another function, `inner`, and returns it. The name `x` is free in `inner`, but it has been resolved in `intervening`, so the same resolution is used in `inner`.

4. Finally, back at the module level, the `f()` statement invokes the function stored in the global variable `f`, which is actually an instance of `inner`. This causes the number 5 to be printed.

Since the global variable `x`, with value 9, is not seen by `outer` or either of its nested functions, we say that the `x` in `outer` *shadows* the global `x`.

In Python, which is an object oriented language, all semantic values are objects, including integers, booleans, closures and modules. As such, they have *attributes*, which are variables associated with an object. Attributes are accessed by the syntax `a.x`, where the object `a` is the *source* of the attribute and `x` is the *name* of the attribute. For general objects (excluding *immediate* objects, such

```
def f():
    x = 1
    def g():
        print x
    g()
    del x
f()
```

Figure 3.3: An IntegerPython program that is not valid in Python

as integers), new attributes can be created by assignment and removed by deletion. In particular, the attributes of a module object are the names in the global namespace of that module. This feature is preserved in IntegerPython, in which modules are the only kind of value that supports attributes.

In order to refer to attributes, the source object must be visible, and the process for making other modules visible within the currently executing module is called *importing*. The `import m as n` statement imports the module named `m` and binds the resulting module object to the local name `n`; when the `as` clause is absent, the local name is the same as the module's name. The first time a module is imported, it is *initialized* by executing its body. For example, if the code in Figure 3.1 were in a module named `freegap`, running the module in Figure 3.2 would produce the following output:

```
5
9 10
10.
```

The `freegap` module, though imported twice, is initialized only once, so the local variables `freegap` and `freegap2` refer to the same module (i.e. global namespace). Thus, `print freegap2.x` prints 10, as opposed to 9.

Finally, the following two IntegerPython statements affect namespaces:

del takes a list of names and attributes and *deletes* their values, so that they revert to an uninitialized state. In a global namespace, this is equivalent to completely removing the name from the namespace, but for local namespaces, the name is left bound to an uninitialized memory location in order to keep it from becoming free. It is a run-time error to delete a name or attribute that has not yet been initialized. Python also forbids (by static analysis) the use of **del** on any name that "occurs as a free variable in a nested [function]" ([33], §6.5), but IntegerPython makes no such restriction. For example, the program in Figure 3.3 is syntactically valid in IntegerPython, but the Python interpreter rejects it with the message, "SyntaxError: can not delete variable 'x' referenced in nested scope." Finally, in both Python and IntegerPython, **del** is considered a binding statement, which effectively prevents it from deleting names from enclosing local namespaces.

```
def f():
    x = 1
    def g():
        global x
        def h():
            print x
        h()
    g()

x = 2
f()
print x
```

Figure 3.4: An IntegerPython program with free and global variables

global is used to specify names that refer to global variables, even if they are apparently bound in the local namespace or an enclosing namespace. In this way, it is possible for functions to assign to or delete global variables. The **global** statement affects the entire namespace in which it occurs, including occurrences of the listed names that precede it. Enclosed namespaces are affected only in the sense that if one of the explicitly global names is free in an (immediately) enclosed namespace, it is made global there also. This is illustrated by the program in Figure 3.4, which prints the number 2 twice because `x` in `h` is made global rather than resolved to the local `x` in `f`. Python forbids (by syntactic analysis) formal parameters from appearing in **global** statements in the corresponding function body. Such appearances are syntactically legal in IntegerPython, but they result in a run-time error when the function is defined.

3.2 The Abstract Syntax of IntegerPython

The abstract syntax of IntegerPython is described in this section. The names of the syntactic constructs are adapted from the "compiler" package of the Python standard library [32], and the introduction of each construct is accompanied by an informal discussion of its meaning. As in the previous chapter, the abstract syntax is presented as a context-free grammar, and we begin by augmenting the notation of grammars with the following constructs:

- For any nonterminal x, a *list* of strings derivable from x, denoted $\widetilde{x}$, is defined as follows:

$$\widetilde{x} \quad ::= \quad nil \mid x :: \widetilde{x},$$

  where nil is an empty list. We also write $[x_1, x_2, \ldots, x_n]$ for the list $x_1 :: x_2 :: \ldots :: x_n :: nil$.

- For any symbols y and z, the *ordered pair* (or simply, *pair*) with first component y and second component z is denoted $\langle y, z \rangle$.

The above constructs exist at the metalinguistic level: they are part of the meta-language used for specifying IntegerPython and should not be confused with Python's list and tuple data types.

The abstract syntax of IntegerPython is divided into the following categories: integers, identifiers, expressions, statements, targets and modules. Integers and identifiers are chosen from the following countably infinite sets of terminal strings, respectively:

$$\begin{array}{rcl} i & \in & \textsc{Integer} \\ n & \in & \textsc{Identifier}. \end{array}$$

Next, *expressions* are defined as the language constructs that are expected to reduce to a meaningful value:

$$\begin{array}{rcll} \textsc{Expression} & \ni & e & ::= & \langle \mathbf{IntLit} \colon i \rangle \\ & & & | & \langle \mathbf{Name} \colon n \rangle \\ & & & | & \langle \mathbf{Getattr} \colon e, n \rangle \\ & & & | & \langle \mathbf{Uop} \colon \upsilon, e \rangle \\ & & & | & \langle \mathbf{Bop} \colon \beta, e, e \rangle \\ & & & | & \langle \mathbf{And} \colon e, \widetilde{e} \rangle \\ & & & | & \langle \mathbf{Or} \colon e, \widetilde{e} \rangle \\ & & & | & \langle \mathbf{Compare} \colon e, \langle \kappa, e \rangle, \widetilde{\langle \kappa, e \rangle} \rangle \\ & & & | & \langle \mathbf{Lambda} \colon \widetilde{n}, e \rangle \\ & & & | & \langle \mathbf{CallFunc} \colon e, \widetilde{e} \rangle. \end{array}$$

The meaning of each case of expression is as follows:

$\langle \mathbf{IntLit} \colon i \rangle$ is an integer literal.

$\langle \mathbf{Name} \colon n \rangle$ retrieves the value of the variable named n, according to the namespace rules of the previous section.

$\langle \mathbf{Getattr} \colon e, n \rangle$ retrieves the value of the attribute with source expression e and name n. The source expression is expected to reduce to a module.

$\langle \mathbf{Uop} \colon \upsilon, e \rangle$ performs the unary operation υ on the operand e. The available unary operations are as follows:

$$\begin{array}{rcll} \textsc{UnaryOp} & \ni & \upsilon & ::= & \mathbf{UnaryPlus} \mid \mathbf{UnaryMinus} \\ & & & | & \mathbf{UnaryInvert} \mid \mathbf{UnaryNot}. \end{array}$$

The $\mathbf{UnaryPlus}$ operation yields "its numeric argument unchanged," while $\mathbf{UnaryMinus}$ yields "the negation of its numeric argument," and $\mathbf{UnaryInvert}$ yields "the bitwise inversion of its [integer] argument" ([33], §5.5). Since two's complement representation is assumed, the inversion of an integer i is $-(i+1)$ by definition [33]. IntegerPython defines these three operators for integers and booleans, which are converted to integers as described below. Finally, the $\mathbf{UnaryNot}$ operation performs boolean negation on its argument, which can be any type and is converted to a boolean according to the rules given in the next section.

$\langle \mathbf{Bop}\colon \beta, e, e\rangle$ performs the binary operation β on its two operand expressions. The available binary operations are as follows:

$$\textsc{BinaryOp} \ni \beta ::= \mathbf{Add} \mid \mathbf{Sub} \mid \mathbf{Mul} \mid \mathbf{Div} \mid \mathbf{Mod}.$$

These are defined only for integer operands (and booleans converted to integers), and they have the obvious arithmetical meaning. Moreover, IntegerPython does not impose size restrictions on integers, so none of these operations is subject to overflow. Division by zero causes a run-time error.

$\langle \mathbf{And}\colon e, \widetilde{e}\rangle$ performs *short-circuit evaluation* of the boolean **and** operation on the nonempty list of expressions $e :: \widetilde{e}$. The expressions are evaluated in order until one evaluates to a false value, which becomes the value of the **And** expression. If none of the listed expressions is false, the value of the last one becomes the value of the **And** expression. The notion of truth and falsehood for values is clarified in the next section.

$\langle \mathbf{Or}\colon e, \widetilde{e}\rangle$ performs short-circuit evaluation of the boolean **or** operation on the nonempty list of expressions $e :: \widetilde{e}$. The expressions are evaluated in order until one evaluates to a true value, which becomes the value of the **Or** expression. If none of the listed expressions is true, the value of the last one becomes the value of the **Or** expression.

$\langle \mathbf{Compare}\colon e, \langle \kappa, e\rangle, \widetilde{\langle \kappa, e\rangle}\rangle$ performs a short-circuit, chained comparison using an initial expression e and a nonempty list of pairs of operators and expressions, $\langle \kappa, e\rangle :: \widetilde{\langle \kappa, e\rangle}$. The available comparison operators are as follows:

$$\textsc{CompareOp} \ni \kappa ::= < \mid <= \mid == \mid != \mid >= \mid >.$$

These have the obvious meaning for integer operands, but IntegerPython defines comparison for all types, in order to agree with Python's practice of defining cross-type comparison "arbitrarily but consistently within one execution of a program" ([33], §5.9). The details of cross-type comparison are explained in Section 3.4.1.

Comparisons may be chained, as in `a < b <= c == d`, which is equivalent to `a < b and b <= c and c == d`, except that each operand is evaluated at most once. The former expression is represented in the abstract syntax as follows:

$$\langle \mathbf{Compare}\colon \langle \mathbf{Name}\colon \texttt{a}\rangle, \langle <, \langle \mathbf{Name}\colon \texttt{b}\rangle\rangle, [\langle <=, \langle \mathbf{Name}\colon \texttt{c}\rangle\rangle, \langle ==, \langle \mathbf{Name}\colon \texttt{d}\rangle\rangle]\rangle.$$

$\langle \mathbf{Lambda}\colon \widetilde{n}, e\rangle$ defines an anonymous function, with formal parameters $\widetilde{n}$, that returns the value of the expression e. The resulting closure is a semantic value, so it can be assigned to a name, invoked immediately with $\langle \mathbf{CallFunc}\colon e, \widetilde{e}\rangle$, or used anywhere else a semantic value is called for. For example, the following program prints the number 3:

```
print (lambda a, b: a + b) (1,2)
```

$\langle$**CallFunc**: $e, \widetilde{e}\rangle$ invokes the closure resulting from the reduction of e, supplying the reductions of the expressions $\widetilde{e}$ as actual parameters.

Boolean literals are notably absent from the list of expressions. Technically, there is no such thing as a boolean literal in Python. The boolean type is a subtype of integers with precisely two instances. Python's `False` and `True` are variables in the "built-in" namespace that initially refer to these instances. However, it is possible by assignment (e.g. `False = 5`) to bind these identifiers to other values. Thus, the only reliable way to get a boolean value is from a comparison or negation. When booleans are used in arithmetic expressions and comparisons, false takes the value 0, and true, 1; only when they are being printed do booleans behave differently from integers ([33], §3.2). Thus, `(5 == 5) + 2` is 3, `(4 == 3) == 0` is true, and so on. This behaviour is preserved in IntegerPython.

A complete IntegerPython program consists of at least one *module*:

$$\text{MODULE} \ni m ::= \langle \mathbf{Module}: s\rangle.$$

The body of a module is a *statement*:

$$\begin{aligned}
\text{STATEMENT} \ni s ::= {} & \langle \mathbf{Stmt}: s, \widetilde{s}\rangle \\
| \ & \langle \mathbf{Pass}\rangle \\
| \ & \langle \mathbf{Discard}: e\rangle \\
| \ & \langle \mathbf{If}: \langle e, s\rangle, \widetilde{\langle e, s\rangle}, s\rangle \\
| \ & \langle \mathbf{While}: e, s, s\rangle \\
| \ & \langle \mathbf{Break}\rangle \\
| \ & \langle \mathbf{Continue}\rangle \\
| \ & \langle \mathbf{Assign}: \widetilde{t}, e\rangle \\
| \ & \langle \mathbf{Global}: \widetilde{n}\rangle \\
| \ & \langle \mathbf{Function}: n, \widetilde{n}, s\rangle \\
| \ & \langle \mathbf{Return}: e\rangle \\
| \ & \langle \mathbf{Import}: \widetilde{\langle n, n\rangle}\rangle \\
| \ & \langle \mathbf{From}: n, \widetilde{\langle n, n\rangle}\rangle \\
| \ & \langle \mathbf{Print}: \widetilde{e}\rangle \\
| \ & \langle \mathbf{Printnl}: \widetilde{e}\rangle.
\end{aligned}$$

In the case of an assignment statement, which is also used to handle **del**, the *target* is defined as follows:

$$\begin{aligned}
\text{TARGET} \ni t ::= {} & \langle \mathbf{AssAttr}: e, n\rangle \\
| \ & \langle \mathbf{AssName}: n\rangle \\
| \ & \langle \mathbf{DelAttr}: e, n\rangle \\
| \ & \langle \mathbf{DelName}: n\rangle.
\end{aligned}$$

The meaning of each case of statement is as follows:

$\langle \mathbf{Stmt}: s, \widetilde{s} \rangle$ represents a nonempty list of statements, $s :: \widetilde{s}$, for sequential execution.

$\langle \mathbf{Pass} \rangle$ is a *nop* (i.e. no operation).

$\langle \mathbf{Discard}: e \rangle$ allows an expression to take the syntactic place of a statement; e is computed, but its value is "discarded." A common use of $\langle \mathbf{Discard}: e \rangle$ is function invocations that ignore the return value, but in an extreme case, it makes `0` a valid IntegerPython program. There is no concrete syntax for $\langle \mathbf{Discard}: e \rangle$; it is generated implicitly by the parser.

$\langle \mathbf{If}: \langle e, s \rangle, \widetilde{\langle e, s \rangle}, s_e \rangle$ represents the **if** statement and consists of a nonempty list of expression-statement pairs, $\langle e, s \rangle :: \widetilde{\langle e, s \rangle}$, called *tests*, and an *else* statement, s_e. An **if** statement is executed by taking the tests in order and evaluating the expressions until one *succeeds* (i.e. reduces to a true value), at which point the corresponding statement is executed and the evaluation of the **if** statement terminates. If no test succeeds, the else statement is executed.

$\langle \mathbf{While}: e, s_b, s_e \rangle$ represents a **while** loop with test expression e, body s_b and else clause s_e. It is executed by repeatedly evaluating e and executing s_b each time e succeeds. When e fails, s_e is executed once and the execution of the loop terminates.

$\langle \mathbf{Break} \rangle$ may appear in a loop body, and if executed, it causes the abrupt termination of the innermost loop that contains it. The test expression is not evaluated again, neither is the rest of the loop body nor the else clause executed.

$\langle \mathbf{Continue} \rangle$ abruptly terminates the execution of the loop body that contains it, but unlike $\langle \mathbf{Break} \rangle$, it does not terminate the loop all together. The loop's test expression is evaluated again, just as it would be if the loop body had been fully executed.

$\langle \mathbf{Assign}: \widetilde{t}, e \rangle$ evaluates e and assigns the resulting value to each target in $\widetilde{t}$. Multiple targets are specified in the concrete syntax by chained assignment, as in `a = b = 5`, which in the abstract syntax is as follows:

$$\langle \mathbf{Assign}: [\langle \mathbf{AssName}: \texttt{a} \rangle, \langle \mathbf{AssName}: \texttt{b} \rangle], \langle \mathbf{IntLit}: 5 \rangle \rangle.$$

The **del** statement is translated into an assignment with a list of $\langle \mathbf{DelName}: n \rangle$ and $\langle \mathbf{DelAttr}: e, n \rangle$ targets, and a trivial right-hand side, such as $\langle \mathbf{IntLit}: 0 \rangle$.

$\langle \mathbf{Global}: \widetilde{n} \rangle$ is the **global** statement described in the previous section.

$\langle \mathbf{Function}: n, \widetilde{n}, s \rangle$ represents the **def** statement and defines a function with name n, parameters $\widetilde{n}$ and body s.

$\langle \mathbf{Return}: e \rangle$ abruptly terminates the most recent function invocation and returns the value of e to the caller.

$\langle \mathbf{Import}: \widetilde{\langle n, n \rangle} \rangle$ is the **import** statement described in the previous section.

$\langle \mathbf{From} : n, \widetilde{\langle n, n \rangle} \rangle$ is an alternative form of importing. The module n is initialized if necessary, and individual names from it are imported into the local namespace. In the pair $\langle n, n \rangle$, the first name is the name of an attribute of the module, and the second is the name in the local namespace to which that attribute's value should be assigned. The module object itself is not assigned a name in the local namespace. Thus, the following code fragments are equivalent, except that the left-hand one does not bind the name `modulename`:

```
from modulename import          | import modulename
    name1 as n1,                | n1 = modulename.name1
    name2,                      | name2 = modulename.name2
    name3 as n3                 | n3 = modulename.n3
```

$\langle \mathbf{Print} : \widetilde{e} \rangle$ prints the expressions in $\widetilde{e}$ in order, separated by spaces, without a terminating line break.

$\langle \mathbf{Printnl} : \widetilde{e} \rangle$ is equivalent to $\langle \mathbf{Print} : \widetilde{e} \rangle$, except that it starts a "new line" after printing the expressions. This is the default interpretation of the **print** statement; to avoid a line break, a trailing comma must be supplied, as in `print 1, 2,`.

3.3 A CEKS Machine for IntegerPython

In this section, we define a CEKS machine for IntegerPython by defining a notation for semantic values, environments, stores, continuations and machine states. The semantic values of IntegerPython are integers, booleans, closures, modules and $\langle$`None`$\rangle$. The $\langle$`None`$\rangle$ value represents the singleton object that is accessible by Python's reserved identifier `None`; it is commonly used in place of an undefined value, such as the return value of a function that does not execute a **return** statement. The semantic values are divided into two categories: those that represent false in a boolean context (ff) and those that represent true (tt). Collectively, they are denoted by the nonterminal u, as follows:

$$
\begin{array}{rrcl}
 & tt & ::= & \langle \texttt{True} \rangle \\
 & & | & i \neq 0 \\
 & & | & \langle \texttt{closure} : s, \widetilde{n}, \widetilde{n}, \epsilon, \epsilon \rangle \\
 & & | & \langle \texttt{module} : \epsilon \rangle \\
 & ff & ::= & \langle \texttt{False} \rangle \\
 & & | & \langle \texttt{None} \rangle \\
 & & | & 0 \\
\textsc{Value} \ni & u & ::= & tt \mid ff,
\end{array}
$$

where the nonterminal ϵ represents a special type of store reference called an *environment handle*.

Recall from Section 3.1, that closures contain "references" to namespaces. It would be impractical to store the global namespace itself in the closure, since many functions may be defined in one global namespace, and the resulting closures would have to be synchronized. Instead, namespaces,

which are represented by environments, are placed in the store and referenced by environment handles. Thus, the environment handle in $\langle \texttt{module} : \epsilon \rangle$ and the second environment handle in $\langle \texttt{closure} : s, \widetilde{n}, \widetilde{n}, \epsilon, \epsilon \rangle$ refer to global namespaces. The first environment handle in a closure refers to an environment in which the free names are bound; this environment could be stored in the closure itself, since it is not shared with any other closure or module, but an environment handle is used for the sake of symmetry.

The notation for environments is similar to that used in Chapter 2, except that, for reasons described below, locations are constructed as environment handle-name pairs rather than chosen from an unspecified infinite set:

$$\begin{array}{rcl} l & ::= & \langle \epsilon, n \rangle \\ E & ::= & \diamond \mid E[n \mapsto l]. \end{array}$$

The empty environment is denoted by $\diamond$, and $E[n \mapsto l]$ denotes the extension of the environment E by adding a binding from n to l. Environment application, $E(n)$, and the domain of an environment, $\mathit{dom}(E)$, are defined as on page 9, using n for names instead of ν. We also define a binary operator, $\oslash$, as the *projection* of an environment onto a list of names:

$$\begin{array}{rcl} E \oslash \mathit{nil} & = & \diamond \\ E \oslash (n :: \widetilde{n}) & = & \begin{cases} (E \oslash \widetilde{n})[n \mapsto E(n)] & \text{if } n \in \mathit{dom}(E) \\ E \oslash \widetilde{n} & \text{if } n \notin \mathit{dom}(E). \end{cases} \end{array}$$

That is, projection yields a new environment that contains only the bindings of those listed names that were bound in the original environment. It is used in Section 3.4.4 to implement the resolution of free names.

The store of an IntegerPython CEKS machine associates not only locations with values, but also environment handles with environments:

$$\begin{array}{rcl} \epsilon & \in & \textsc{EnvironmentHandle} \\ r & ::= & l \mid \epsilon \\ S & ::= & \circ \mid S[l] \mid S[l \mapsto u] \mid S[\epsilon \mapsto E]. \end{array}$$

The EnvironmentHandle set can be any countably infinite set with a total order, $\leq$. The $\circ$ symbol denotes an empty store, $S[l]$ denotes the extension of a store by an uninitialized location, $S[l \mapsto u]$ denotes the extension of a store by a location-value pair, and $S[\epsilon \mapsto E]$ denotes the extension of a store by an environment handle-environment pair. Collectively, locations and environment handles are known as *references* (r). Applying a store to a reference, denoted $S(r)$, is known as *dereferencing* and is defined as follows:

$$\begin{array}{rcll} S[l](r) & = & S(r) & \text{if } l \neq r \\ S[l \mapsto u](r) & = & \begin{cases} u & \text{if } l = r \\ S(r) & \text{if } l \neq r \end{cases} \\ S[\epsilon \mapsto E](r) & = & \begin{cases} E & \text{if } \epsilon = r \\ S(r) & \text{if } \epsilon \neq r. \end{cases} \end{array}$$

Moreover, the following equations define the *domain* of a store:

$$\begin{aligned} \mathit{dom}(\circ) &= \emptyset \\ \mathit{dom}(S[l]) &= \{l\} \cup \mathit{dom}(S) \\ \mathit{dom}(S[l \mapsto u]) &= \{l\} \cup \mathit{dom}(S) \\ \mathit{dom}(S[\epsilon \mapsto E]) &= \{\epsilon\} \cup \mathit{dom}(S). \end{aligned}$$

Since a location may belong to the domain of a store even when no value is associated with it (i.e. the $S[l]$ case), we define the set of *initialized* locations of a store as follows:

$$\begin{aligned} \mathit{init}(\circ) &= \emptyset \\ \mathit{init}(S[l]) &= \mathit{init}(S) - \{l\} \\ \mathit{init}(S[l \mapsto u]) &= \{l\} \cup \mathit{init}(S) \\ \mathit{init}(S[\epsilon \mapsto E]) &= \mathit{init}(S). \end{aligned}$$

Lastly, the following shorthand notation creates a new variable by binding the name n to the location $\langle \epsilon, n \rangle$ in the environment $S(\epsilon)$ and adding the value and the new environment to the store:

$$S[\epsilon, n, u] \;=\; S[\langle \epsilon, n \rangle \mapsto u][\epsilon \mapsto S(\epsilon)[n \mapsto \langle \epsilon, n \rangle]].$$

This notation exploits the fact that a namespace-name pair is sufficient to identify any program variable. That is, adding locations to a store exclusively in this way guarantees that each new location $\langle \epsilon, n \rangle \notin \mathit{dom}(S)$. In fact, this practice makes environments redundant in many cases, because if $S(\epsilon)(n) = \langle \epsilon', n' \rangle$, then $n = n'$ in general (see Appendix C), and in the absence of aliases generated by the resolution of free names in Section 3.4.5, $\epsilon = \epsilon'$, as well. In this way, the locations of global variables, function parameters, etc. can be fully predicted. Moreover, every name in a given environment binds to a distinct location, since the variable's name is part of the location. The authors have found these properties to simplify the task of program verification (Section 4.2) by facilitating the division of programs into smaller sub-programs for analysis.

The continuations (K) of the IntegerPython CEKS machine are defined by the following grammar:

$$\begin{array}{rcll} K & ::= & L \mid \langle \mathsf{WhileRun}\colon e, s, s \rangle \mid \langle \mathsf{Invocation} \rangle & \\ L & ::= & \langle \mathsf{Assign}\colon \widetilde{t} \rangle & \mid \langle \mathsf{AssAttr}\colon n, u \rangle \\ & \mid & \langle \mathsf{BopLeft}\colon \beta, e \rangle & \mid \langle \mathsf{BopRight}\colon \beta, u \rangle \\ & \mid & \langle \mathsf{CallFunc}\colon \widetilde{e} \rangle & \mid \langle \mathsf{CallFunc2}\colon \widetilde{n}, \widetilde{e}, s, \epsilon, \epsilon \rangle \\ & \mid & \langle \mathsf{Compare}\colon \langle \kappa, e \rangle, \widetilde{\langle \kappa, e \rangle} \rangle & \mid \langle \mathsf{Compare2}\colon u, \kappa, \widetilde{\langle \kappa, e \rangle} \rangle \\ & \mid & \langle \mathsf{DelAttr}\colon n, u \rangle & \mid \langle \mathsf{Discard} \rangle \\ & \mid & \langle \mathsf{From}\colon n, \widetilde{\langle n, n \rangle} \rangle & \mid \langle \mathsf{GetAttr}\colon n \rangle \\ & \mid & \langle \mathsf{If}\colon s, \widetilde{\langle e, s \rangle}, s \rangle & \mid \langle \mathsf{Import}\colon \widetilde{\langle n, n \rangle} \rangle \\ & \mid & \langle \mathsf{Import2}\colon n, \epsilon \rangle & \mid \langle \mathsf{ParamBind}\colon n, \epsilon \rangle \\ & \mid & \langle \mathsf{Print}\colon \widetilde{e} \rangle & \mid \langle \mathsf{Println}\colon \widetilde{e} \rangle \\ & \mid & \langle \mathsf{RestoreEnv}\colon \epsilon, \epsilon \rangle & \mid \langle \mathsf{And}\colon \widetilde{e} \rangle \\ & \mid & \langle \mathsf{Or}\colon \widetilde{e} \rangle & \mid \langle \mathsf{Stmt}\colon \widetilde{s} \rangle \\ & \mid & \langle \mathsf{Uop}\colon u \rangle & \mid \langle \mathsf{WhileTest}\colon e, s, s \rangle. \end{array}$$

These continuations are not nested, as the toy language continuations were. Instead, stacking is achieved by storing a list of configurations, $\widetilde{K}$, in the K register of the machine, where the first element in the list is the top of the stack. The meaning of each continuation and the reason for the distinction of *local continuations* (L) are clarified in the next section.

With respect to machine states, it is no longer practical to store all the names that may be bound at a program point in one environment, because module attributes provide random access to global namespaces. Therefore, a second environment register is used for the global environment. Also, the environment registers no longer contain the environments themselves, but only environment handles, so that the actual environment is found by dereferencing the register's value in the store. Therefore, machine states are defined as follows:

$$\begin{array}{rcl} M & ::= & \langle m, \epsilon, \epsilon, \widetilde{K}, S \rangle \\ & | & \langle s, \epsilon, \epsilon, \widetilde{K}, S \rangle \\ & | & \langle e, \epsilon, \epsilon, \widetilde{K}, S \rangle \\ & | & \langle u, \epsilon, \epsilon, \widetilde{K}, S \rangle. \end{array}$$

3.4 Execution

In this section, we describe the transition rules of the IntegerPython CEKS machine using the $\longrightarrow$ notation introduced in Section 2.1. The discussion is divided into subsections, each describing the rules related to a particular language feature. The order of the topics is as follows: expressions (3.4.1), container statements (3.4.2), assignment and deletion (3.4.3), function definition (3.4.4), function invocation (3.4.5), returning from functions (3.4.6), module importing (3.4.7), loops (3.4.8), conditional statements (3.4.9), printing (3.4.10) and miscellanea (3.4.11).

3.4.1 Expressions

The transition rules for evaluating expressions are numerous but generally simple. The discussion is ordered as follows: literals, names and attributes, unary and binary operations, short-circuit operations and comparisons. (Lambda expressions and function invocations are deferred to Sections 3.4.4 and 3.4.5 respectively.) By convention, the rules that refer to partial functions do not apply where the function is undefined. In this way, run-time errors are indicated by allowing the machine to get stuck (i.e. to have no applicable rule).

Literals, names and attributes are reduced according to the rules in Figure 3.5. The R_{intlit} rule trivially reduces integer literals to integer values. In the reduction of names by R_{name}, there are three cases: The first case applies to the reserved identifier `None`, which always stores the value $\langle$`None`$\rangle$. The second case applies to a name that is locally bound to an initialized location, where applying the store to the bound location yields the variable's value. The third case is similar to the second and applies to a name that is bound globally but not locally ($n \in dom(E_G)$ is implicit). R_{name} does not apply if n is bound to an uninitialized location, or if it is not bound at all.

In attribute retrieval, the source expression is scheduled by $R_{getattr}$, which also remembers the attribute name in a $\langle \mathsf{GetAttr} : n \rangle$ continuation. If the source expression reduces to a module, $C_{getattr}$

$$R_{intlit}: \quad \langle\langle \mathbf{IntLit}: i\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle i, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

$$R_{name}: \quad \langle\langle \mathbf{Name}: n\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow$$
$$\begin{cases} \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle & \text{if } n = \texttt{None} \\ \langle S(E_L(n)), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle & \text{if } n \in \mathit{dom}(E_L) \text{ and } E_L(n) \in \mathit{init}(S) \\ \langle S(E_G(n)), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle & \text{if } n \notin \mathit{dom}(E_L) \text{ and } E_G(n) \in \mathit{init}(S) \end{cases}$$
where $E_L = S(\epsilon_L)$ and $E_G = S(\epsilon_G)$

$$R_{getattr}: \quad \langle\langle \mathbf{Getattr}: e, n\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{GetAttr}: n\rangle :: \widetilde{K}, S\rangle$$
$$C_{getattr}: \quad \langle\langle \texttt{module} : \epsilon\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{GetAttr}: n\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle S(S(\epsilon)(n)), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

Figure 3.5: Transition rules for literals, names and attributes

dereferences the binding of the name in the module's global namespace, $S(\epsilon)$, to give the value of the attribute. If $S(\epsilon)$, $S(\epsilon)(n)$ or $S(S(\epsilon)(n))$ is undefined, $C_{getattr}$ does not apply.

The transition rules for unary and binary operations are given in Figure 3.6. R_{uop} schedules the expression to which the unary operator υ is to be applied and remembers the operator in the $\langle \mathsf{Uop}: \upsilon\rangle$ continuation. C_{uop} applies the operator according to the partial function U : UNARYOP$\times$ VALUE $\rightarrow$ VALUE, which is defined as follows:

$$\begin{aligned} U(\mathbf{UnaryPlus}, i) &= i \\ U(\mathbf{UnaryMinus}, i) &= -i \\ U(\mathbf{UnaryInvert}, i) &= -(i+1) \\ U(\mathbf{UnaryNot}, \mathit{ff}) &= \langle \texttt{True}\rangle \\ U(\mathbf{UnaryNot}, \mathit{tt}) &= \langle \texttt{False}\rangle \\ U(\upsilon, \langle \texttt{False}\rangle) &= U(\upsilon, 0) \\ U(\upsilon, \langle \texttt{True}\rangle) &= U(\upsilon, 1). \end{aligned}$$

The $\mathbf{UnaryNot}$ operation is defined for all values, since any value may be converted to a boolean; the others are defined only for integers and booleans converted to integers.

With respect to binary operations, R_{bop} schedules the left operand and remembers the operator and the right operand. $C_{bopleft}$ schedules the right operand and remembers the operator and left operand's value. Finally, $C_{bopright}$ computes the result using the partial function B : BINARYOP $\times$

$$
\begin{aligned}
R_{uop}: &\quad \langle\langle \mathbf{Uop}: \upsilon, e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Uop}: \upsilon\rangle :: \widetilde{K}, S\rangle \\
C_{uop}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Uop}: \upsilon\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle U(\upsilon, u), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
\\
R_{bop}: &\quad \langle\langle \mathbf{Bop}: \beta, e_L, e_R\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e_L, \epsilon_L, \epsilon_G, \langle \mathsf{BopLeft}: \beta, e_R\rangle :: \widetilde{K}, S\rangle \\
C_{bopleft}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{BopLeft}: \beta, e_R\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e_R, \epsilon_L, \epsilon_G, \langle \mathsf{BopRight}: \beta, u\rangle :: \widetilde{K}, S\rangle \\
C_{bopright}: &\quad \langle u_R, \epsilon_L, \epsilon_G, \langle \mathsf{BopRight}: \beta, u_L\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle B(\beta, u_L, u_R), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.6: Transition rules for unary and binary operations

VALUE $\times$ VALUE $\rightarrow$ VALUE, which is defined as follows:

$$
\begin{aligned}
B(\mathbf{AddOp}, i_L, i_R) &= i_L + i_R \\
B(\mathbf{SubOp}, i_L, i_R) &= i_L - i_R \\
B(\mathbf{MulOp}, i_L, i_R) &= i_L i_R \\
B(\mathbf{DivOp}, i_L, i_R) &= \left\lfloor \frac{i_L}{i_R} \right\rfloor \text{ if } i_R \neq 0 \\
B(\mathbf{ModOp}, i_L, i_R) &= i_L - i_R \left\lfloor \frac{i_L}{i_R} \right\rfloor \text{ if } i_R \neq 0 \\
B(\beta, \langle \texttt{False} \rangle, u) &= B(\beta, 0, u) \\
B(\beta, \langle \texttt{True} \rangle, u) &= B(\beta, 1, u) \\
B(\beta, u, \langle \texttt{False} \rangle) &= B(\beta, u, 0) \\
B(\beta, u, \langle \texttt{True} \rangle) &= B(\beta, u, 1),
\end{aligned}
$$

where $\left\lfloor \frac{i_L}{i_R} \right\rfloor$ refers to *floor division*, which results in the greatest integer that is less than or equal to the rational number $\frac{i_L}{i_R}$. For example, $\left\lfloor \frac{2}{5} \right\rfloor = 0$ and $\left\lfloor \frac{-10}{4} \right\rfloor = -3$. It is not clear that this definition of division and modulus is exactly that used by Python, but it does satisfy the following identity claimed for Python: "`x == (x/y) * y + (x%y)`" ([33], §5.6). That is, $\left\lfloor \frac{i_L}{i_R} \right\rfloor i_R + (i_L - i_R \left\lfloor \frac{i_L}{i_R} \right\rfloor) = i_L$, where $i_R \neq 0$.

The short-circuit boolean operations, $\langle \mathbf{And}: e, \widetilde{e}\rangle$ and $\langle \mathbf{Or}: e, \widetilde{e}\rangle$, are evaluated by the rules in Figure 3.7. Recall that $\langle \mathbf{And}: e, \widetilde{e}\rangle$ should yield the value of the first false expression in $e :: \widetilde{e}$, and $\langle \mathbf{Or}: e, \widetilde{e}\rangle$, the first true expression. In both cases, if no expressions are found with the respective truth values, the value of the last expression in the list is yielded. R_{scor} evaluates $\langle \mathbf{Or}: e, \widetilde{e}\rangle$ by scheduling the first expression, e, and remembering the rest in the $\langle \mathsf{Or}: \widetilde{e}\rangle$ continuation. If that expression reduces to a false value, C_{scorf} schedules the next expression, if any remain; the continuation is popped either by $C_{scornil}$ when no expressions remain, or by C_{scort} when a true expression has been found. The evaluation of $\langle \mathbf{And}: e, \widetilde{e}\rangle$ is analogous. The use of the nonempty list pattern, $e :: \widetilde{e}$, in C_{scort} ond C_{scandf} is not necessary for correct evaluation, but it prevents overlap

$$
\begin{aligned}
R_{scor}: &\quad \langle\langle \mathbf{Or}: e, \widetilde{e}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Or}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{scorf}: &\quad \langle \mathit{ff}, \epsilon_L, \epsilon_G, \langle \mathsf{Or}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Or}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{scort}: &\quad \langle \mathit{tt}, \epsilon_L, \epsilon_G, \langle \mathsf{Or}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle \mathit{tt}, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
C_{scornil}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Or}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
\\
R_{scand}: &\quad \langle\langle \mathbf{And}: e, \widetilde{e}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{And}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{scandt}: &\quad \langle \mathit{tt}, \epsilon_L, \epsilon_G, \langle \mathsf{And}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{And}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{scandf}: &\quad \langle \mathit{ff}, \epsilon_L, \epsilon_G, \langle \mathsf{And}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle \mathit{ff}, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
C_{scandnil}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{And}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.7: Transition rules for short-circuit expressions

with the patterns of $C_{scornil}$ and $C_{scandnil}$, thus preserving the overall distinctness of the patterns of the rules. This distinctness is necessary for the admission of the transition rules as Isabelle/HOL functions in Appendix B, and it ensures that the CEKS machine is deterministic.

Comparison expressions are similar to short-circuit expressions, in that IntegerPython allows chained comparisons such as x <= y <= z, which is logically equivalent to (x <= y) **and** (y <= z). However, IntegerPython follows Python's proviso that y is "evaluated only once [in such an expression]" ([33], §5.9), which prevents chained comparisons from being translated in this way. Instead, comparisons are evaluated by the transition rules in Figure 3.8. $R_{compare}$ schedules the initial expression e and remembers the operator-expression pairs; $C_{compare}$ uses the value of e to prime the Compare2 continuation, so that the repetitive evaluation of the comparisons can be performed by $C_{compare2}$.

The truth or falsehood of each comparison is decided by the function

$$C : \textsc{Value} \times \textsc{CompareOp} \times \textsc{Value} \to \{\mathtt{T}, \mathtt{F}\},$$

where T and F represent logical truth and falsehood respectively. In IntegerPython, integers and booleans are compared arithmetically, according to Python's practice of comparing "numbers" arithmetically ([33], §5.9). Therefore, C is defined for integers and booleans as follows:

$$
\begin{aligned}
C(i_L, <, i_R) &= (i_L < i_R) \\
C(i_L, ==, i_R) &= (i_L = i_R) \\
C(u, \kappa, \langle \mathtt{False}\rangle) &= C(u, \kappa, 0) \\
C(u, \kappa, \langle \mathtt{True}\rangle) &= C(u, \kappa, 1).
\end{aligned}
$$

Otherwise, most of Python's built-in types, including modules and closures, use *referential equality*, so that objects "compare unequal unless they are the same object" [33]. IntegerPython simulates this with *deep equality*, so that, with the above exceptions, $C(u_L, ==, u_R) = \mathtt{T}$ if and only

$$R_{compare}: \quad \langle\langle \textbf{Compare}: e, \langle \kappa, e\rangle, \widetilde{\langle \kappa, e\rangle}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \textbf{Compare}: \langle \kappa, e\rangle, \widetilde{\langle \kappa, e\rangle}\rangle :: \widetilde{K}, S\rangle$$

$$C_{compare}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \textbf{Compare}: \langle \kappa, e\rangle, \widetilde{\langle \kappa, e\rangle}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \textbf{Compare2}: u, \kappa, \widetilde{\langle \kappa, e\rangle}\rangle :: \widetilde{K}, S\rangle$$

$$C_{compare2}: \quad \langle u_R, \epsilon_L, \epsilon_G, \langle \textbf{Compare2}: u_L, \kappa, \langle \kappa', e\rangle :: \widetilde{\langle \kappa, e\rangle}\rangle :: \widetilde{K}, S\rangle \longrightarrow \begin{cases} \langle e, \epsilon_L, \epsilon_G, \langle \textbf{Compare2}: u_R, \kappa', \widetilde{\langle \kappa, e\rangle}\rangle :: \widetilde{K}, S\rangle & \text{if } C(u_L, \kappa, u_R) = \texttt{T} \\ \langle\langle \texttt{False}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle & \text{if } C(u_L, \kappa, u_R) = \texttt{F} \end{cases}$$

$$C_{compare2nil}: \quad \langle u_R, \epsilon_L, \epsilon_G, \langle \textbf{Compare2}: u_L, \kappa, nil\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle C(u_L, \kappa, u_R), \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

Figure 3.8: Transition rules for comparison expressions

if u_L and u_R are identical. In fact, this simulation is accurate for the types supported by IntegerPython. Clearly, objects that are referentially equal are deeply equal. Moreover, for a singleton object such as `None`, referential and deep equality are equivalent. For modules (closures), the global (respectively, resolution) environment handle may be interpreted as an object identity, so that deep equality of modules and closures implies that they are indeed "the same object." Therefore, it is also reasonable to order non-numeric values as follows:

$$\begin{aligned} C(\langle \texttt{None}\rangle, <, u) &= u \neq \langle \texttt{None}\rangle \\ C(u, <, \langle \texttt{None}\rangle) &= \texttt{F} \\ C(\langle \texttt{closure} : \cdot, \cdot, \cdot, \epsilon_L, \cdot\rangle, <, \langle \texttt{closure} : \cdot, \cdot, \cdot, \epsilon_R, \cdot\rangle) &= \epsilon_L \leq \epsilon_R \wedge \epsilon_L \neq \epsilon_R \\ C(\langle \texttt{closure} : \cdot, \cdot, \cdot, \cdot, \cdot\rangle, <, u) &= \texttt{T} \\ C(\langle \texttt{module} : \cdot\rangle, <, \langle \texttt{closure} : \cdot, \cdot, \cdot, \cdot, \cdot\rangle) &= \texttt{F} \\ C(\langle \texttt{module} : \epsilon_L\rangle, <, \langle \texttt{module} : \epsilon_R\rangle) &= \epsilon_L \leq \epsilon_R \wedge \epsilon_L \neq \epsilon_R \\ C(\langle \texttt{module} : \cdot\rangle, <, u) &= \texttt{T} \\ C(i, <, u) &= \texttt{F}, \end{aligned}$$

where overlap is resolved by sequential precedence. This ordering is consistent, but it may not agree with Python's ordering in all cases, since the latter is also "arbitrary" [33]. Finally, the remaining comparison operators are defined in terms of $<$ and $==$:

$$\begin{aligned} C(u_L, <=, u_R) &= C(u_L, <, u_R) \vee C(u_L, ==, u_R) \\ C(u_L, >, u_R) &= \neg C(u_R, <=, u_L) \\ C(u_L, >=, u_R) &= \neg C(u_L, <, u_R). \end{aligned}$$

$$
\begin{aligned}
R_{\mathit{stmt}}: &\quad \langle\langle \mathbf{Stmt}: s, \widetilde{s}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle s, \epsilon_L, \epsilon_G, \langle \mathsf{Stmt}: \widetilde{s}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{stmt}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Stmt}: s :: \widetilde{s}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle s, \epsilon_L, \epsilon_G, \langle \mathsf{Stmt}: \widetilde{s}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{stmtnil}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Stmt}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_{\mathit{discard}}: &\quad \langle\langle \mathbf{Discard}: e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Discard}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{discard}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Discard}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.9: Transition rules for container statements

$$
\begin{aligned}
R_{\mathit{assign}}: &\quad \langle\langle \mathbf{Assign}: \widetilde{t}, e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \widetilde{t}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{assignnil}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.10: Transition rules for assignment in general

3.4.2 Container statements

There are two IntegerPython statements that serve only to contain other syntactic constructs: $\langle \mathbf{Stmt}: s, \widetilde{s}\rangle$ contains a sequence of statements and $\langle \mathbf{Discard}: e\rangle$ contains an expression. These *container statements* are reduced according to the rules in Figure 3.9. The $\langle \mathbf{Stmt}: s, \widetilde{s}\rangle$ statement specifies one or more statements for sequential execution, so R_{stmt} schedules the first statement and remembers the rest in the $\langle \mathsf{Stmt}: \widetilde{s}\rangle$ continuation. Once the current statement has been reduced to a value, C_{stmt} schedules the next; if none remain, C_{stmtnil} pops the continuation. Moreover, C_{stmtnil} establishes the convention of reducing statements to $\langle \mathtt{None}\rangle$. The exact value to which statements are reduced ought not to matter in practice, since only expressions are expected to reduce to a meaningful value. (The exception is $\langle \mathbf{Return}: e\rangle$, which circumvents C_{stmtnil} by the rules in Section 3.4.6.) However, since it is not easy to prove that the choice of value never affects the subsequent execution, care is taken to ensure that *artifacts* from the reduction of a statement do not reach the statement's context. In keeping with this convention, R_{discard} reduces $\langle \mathbf{Discard}: e\rangle$ by scheduling e and pushing a continuation that discards the value of e by changing it to $\langle \mathtt{None}\rangle$ (C_{discard}).

3.4.3 Assignment and deletion

An $\langle \mathbf{Assign}: \widetilde{t}, e\rangle$ statement consists of an expression that is to be assigned to a list of targets. Individually, the targets may represent either ordinary assignment to a name or an attribute, or the deletion of a name or an attribute. The first and last transition rules that apply in the evaluation of an assignment statement are shown in Figure 3.10, where R_{assign} schedules the expression to be assigned and remembers the targets, and $C_{\mathit{assignnil}}$ pops the continuation after all the assignments have been performed. The rest of this subsection describes the semantics of assigning to each type

$$C_{nassign}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \langle \mathbf{AssName}: n\rangle :: \widetilde{t}\rangle :: \widetilde{K}, S\rangle \longrightarrow$$
$$\langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \widetilde{t}\rangle :: \widetilde{K}, S'\rangle$$
$$\text{if } n \neq \texttt{None}, \text{ where } S' = \begin{cases} S[S(\epsilon_L)(n) \mapsto u] & \text{if } n \in \mathit{dom}(S(\epsilon_L)) \\ S[S(\epsilon_G)(n) \mapsto u] & \text{if } n \notin \mathit{dom}(S(\epsilon_L)) \wedge n \in \mathit{dom}(S(\epsilon_G)) \\ S[\epsilon_G, n, u] & \text{if } n \notin \mathit{dom}(S(\epsilon_L)) \cup \mathit{dom}(S(\epsilon_G)) \end{cases}$$

$$C_{assattr}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \langle \mathbf{AssAttr}: e, n\rangle :: \widetilde{t}\rangle :: \widetilde{K}, S\rangle \longrightarrow$$
$$\langle e, \epsilon_L, \epsilon_G, \langle \mathsf{AssAttr}: n, u\rangle :: \langle \mathsf{Assign}: \widetilde{t}\rangle :: \widetilde{K}, S\rangle$$
$$\text{if } n \neq \texttt{None}$$

$$C_{assattr2}: \quad \langle \langle \texttt{module} : \epsilon\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{AssAttr}: n, u\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_L, \epsilon_G, \widetilde{K}, S'\rangle$$
$$\text{where } S' = \begin{cases} S[S(\epsilon)(n) \mapsto u] & \text{if } n \in \mathit{dom}(S(\epsilon)) \\ S[\epsilon, n, u] & \text{if } n \notin \mathit{dom}(S(\epsilon)) \end{cases}$$

Figure 3.11: Transition rules for name and attribute assignment

of target.

Assignment to names is performed by the $C_{nassign}$ rule in Figure 3.11. Assignment to the reserved name `None` is forbidden. Otherwise, $C_{nassign}$ assigns a value to an $\langle \mathbf{AssName}: n\rangle$ target by first looking up the binding of n in the local environment. If it exists, the corresponding location is used to perform the assignment. If not, the second case of $C_{nassign}$ tries to use a binding from the global environment. If the name is not bound by either environment, the third case of $C_{nassign}$ binds it to the location $\langle \epsilon_G, n\rangle$, in the *global* environment. In this way, names may be made explicitly global by leaving them out of the local environment. Locally bound names, which occur only in functions, are bound to an uninitialized store location when the function is invoked, so that by the time the assignment statement is reached, a binding already exists in the local environment.

In the case of attribute assignment, $C_{assattr}$ (Figure 3.11) determines the environment to be used by scheduling the source expression of the attribute, which should reduce to a $\langle \texttt{module} : \epsilon\rangle$ value. The name of the attribute, which must not be the reserved name `None`, and the value to be assigned are remembered in the $\langle \mathsf{AssAttr}: n, u\rangle$ continuation; the assignment is performed by $C_{assattr2}$, which reuses an existing binding if possible. Both $C_{assattr2}$ and $C_{nassign}$ leave the value to be assigned in the C register in case there are more targets.

Deletion of names and attributes is performed by the transition rules in Figure 3.12, which find the target's location similarly to the related rules for assignment targets. Since it is an error to delete an uninitialized target, both $C_{delname}$ and $C_{delattr2}$ are conditional on the target location's being in $\mathit{init}(S)$. They are also implicitly conditional on the existence of a binding for the target, due to the unchecked use of $S(\epsilon_G)(n)$, etc. The actual deletion is performed by reverting the target's location

$$C_{delname}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \langle \mathbf{DelName}: n\rangle :: \widetilde{t}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \widetilde{t}\rangle :: \widetilde{K}, S[l]\rangle$$
$$\text{if } n \neq \texttt{None} \text{ and } l \in \mathit{init}(S), \text{ where } l = \begin{cases} S(\epsilon_L)(n) & \text{if } n \in \mathit{dom}(S(\epsilon_L)) \\ S(\epsilon_G)(n) & \text{if } n \notin \mathit{dom}(S(\epsilon_L)) \end{cases}$$

$$C_{delattr}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Assign}: \langle \mathbf{DelAttr}: e, n\rangle :: \widetilde{t}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{DelAttr}: n, u\rangle :: \langle \mathsf{Assign}: \widetilde{t}\rangle :: \widetilde{K}, S\rangle$$
$$\text{if } n \neq \texttt{None}$$

$$C_{delattr2}: \quad \langle\langle \texttt{module} : \epsilon\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{DelAttr}: n, u\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_L, \epsilon_G, \widetilde{K}, S[S(\epsilon)(n)]\rangle$$
$$\text{if } S(\epsilon)(n) \in \mathit{init}(S)$$

Figure 3.12: Transition rules for deletion

to an uninitialized state by the $S[l]$ form of store extension.

3.4.4 Function definition

Recall that functions can be defined in two ways: named functions are defined by the **def** statement, and anonymous functions are defined by lambda expressions. In both cases, the result is a closure, which is either assigned to a name or used immediately in an expression. The transition rules for reducing each kind of function definition are given in Figure 3.13, with reference to the following functions: *globalnames*(s), *boundnames*$(s, \widetilde{n})$ and *freenames*$(s, \widetilde{n})$. The *globalnames*(s) function yields all names that occur in **global** statements within the given statement, s; *boundnames*$(s, \widetilde{n})$ yields all names that are bound by binding statements within s, excluding the names in $\widetilde{n}$; *freenames*$(s, \widetilde{n})$ yields all names that occur in s but are not bound or explicitly global, excluding the names in $\widetilde{n}$. These are defined by their respective Isabelle implementations in Appendix B.

The $R_{function}$ rule reduces a function definition with name n, formal parameters $\widetilde{n}$ and body s as follows:

- A closure is computed and placed in the C register.
- An environment, referenced by the fresh environment handle ϵ, is created to store the resolutions of the function's free names, as given by *freenames*$(s, \widetilde{n})$. These are found by projecting the current local environment onto the list of free names. In this way, the resolved names become aliases of the corresponding names in the enclosing local namespace, in that they are bound to the same store location. Thus, assignments in the enclosing namespace affect

$$R_{function}: \quad \langle\langle \mathbf{Function}: n, \widetilde{n}, s\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow$$
$$\langle\langle \texttt{closure}: s, \widetilde{n}, \mathit{boundnames}(s, \widetilde{n}), \epsilon, \epsilon_G\rangle, \epsilon_L, \epsilon_G,$$
$$\langle \mathsf{Assign}: [\langle \mathbf{AssName}: n\rangle]\rangle :: \widetilde{K}, S[\epsilon \mapsto S(\epsilon_L) \oslash \mathit{freenames}(s, \widetilde{n})]\rangle$$
if $\mathit{globalnames}(s) \sqcap \widetilde{n} = nil$, where $\epsilon \notin \mathit{dom}(S)$.

$$R_{lambda}: \quad \langle\langle \mathbf{Lambda}: \widetilde{n}, e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow$$
$$\langle\langle \texttt{closure}: \langle \mathbf{Return}: e\rangle, \widetilde{n}, nil, \epsilon_L, \epsilon_G\rangle, \epsilon_L, \epsilon_G, \widetilde{K},$$
$$S[\epsilon \mapsto S(\epsilon_L) \oslash \mathit{freenames}(\langle \mathbf{Return}: e\rangle, \widetilde{n})]\rangle$$
where $\epsilon \notin \mathit{dom}(S)$

Figure 3.13: Transition rules for function definition

the value that is seen by the function, which agrees with Python's behaviour. The function itself cannot change the value of a resolved variable, since any assignment to it prevents it from being free in the first place.

- The assignment of the closure to the name n is performed by reusing the transition rules for assignment. Specifically, pushing the $\langle \mathsf{Assign}: [\langle \mathbf{AssName}: n\rangle]\rangle$ continuation causes the closure in C to be assigned to the name n and accounts for all the special cases of assignment, including explicitly global names.

The $R_{function}$ rule applies only if no name appears as both a formal parameter and a global, as expressed by the constraint $\mathit{globalnames}(s) \sqcap \widetilde{n} = nil$, where $\sqcap$ represents a list intersection operation that is analogous to set intersection. This constraint satisfies the stipulation, in Section 3.1, that the appearance of a formal parameter in a **global** statement causes a run-time error when the function is defined.

Lambda forms are reduced similarly to functions, except that no assignment is performed; R_{lambda} creates a closure in C but does not change K. Since lambda forms are intended to compute expressions, the function body is simply $\langle \mathbf{Return}: e\rangle$. It follows that there are no bound names, and no explicitly global names. Free names are resolved from the current local environment, although the absence of explicitly global names means that projection is not strictly necessary. That is, it would be equivalent to define $S' = S[\epsilon \mapsto S(\epsilon_L)]$, except that this wastes space.

3.4.5 Function invocation

With reference to the transition rules in Figure 3.14, the invocation of a function (i.e. the reduction of a $\langle \mathbf{CallFunc}: e, \widetilde{e}\rangle$ expression) proceeds as follows:

1. $R_{callfunc}$ schedules the function expression e, which should reduce to a closure. The actual parameters are remembered in the $\langle \mathsf{CallFunc}: \widetilde{e}\rangle$ continuation.

$$R_{callfunc}: \quad \langle\langle \mathbf{CallFunc}: e, \widetilde{e}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{CallFunc}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle$$

$$\begin{aligned} C_{callfunc}: \quad & \langle\langle \texttt{closure} : s, \widetilde{n}_p, \widetilde{n}_b, \epsilon_r, \epsilon_g\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{CallFunc}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \\ & \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{CallFunc2}: \widetilde{n}_p, \widetilde{e}, s, \epsilon_t, \epsilon_g\rangle :: \widetilde{K}, \mathbb{L}(S, S(\epsilon_r), \epsilon_t, \widetilde{n}_b)\rangle \\ & \text{where } \epsilon_t \notin \mathit{dom}(S) \end{aligned}$$

$$\begin{aligned} C_{callfunc2}: \quad & \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{CallFunc2}: n :: \widetilde{n}, e :: \widetilde{e}, s, \epsilon_t, \epsilon_g\rangle :: \widetilde{K}, S\rangle \longrightarrow \\ & \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{ParamBind}: n, \epsilon_t\rangle :: \langle \mathsf{CallFunc2}: \widetilde{n}, \widetilde{e}, s, \epsilon_t, \epsilon_g\rangle :: \widetilde{K}, S\rangle \end{aligned}$$

$$C_{parambind}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{ParamBind}: n, \epsilon\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S[\epsilon, n, u]\rangle$$

$$\begin{aligned} C_{callfunc2nil}: \quad & \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{CallFunc2}: nil, nil, s, \epsilon_t, \epsilon_g\rangle :: \widetilde{K}, S\rangle \longrightarrow \\ & \langle s, \epsilon_t, \epsilon_g, \langle \mathsf{Invocation}\rangle :: \langle \mathsf{RestoreEnv}: \epsilon_L, \epsilon_G\rangle :: \widetilde{K}, S\rangle \end{aligned}$$

Figure 3.14: Transition rules for function invocation

2. $C_{callfunc}$ proceeds with the invocation if the function expression reduced to a closure. A copy of the closure's resolution environment, referenced by the fresh environment handle ϵ_t, is created to serve as the invocation's local environment. Next, each of the function's bound names is bound in the new environment to an uninitialized store location. This distinguishes locally bound names, so that the use of a bound name before its binding statement may be recognized as an error and global names may be implemented simply by leaving them out of the local environment. The binding is performed by the function $\mathbb{L}$, which is defined as follows:

$$\begin{aligned} \mathbb{L}(S, E, \epsilon, nil) &= S[\epsilon \mapsto E] \\ \mathbb{L}(S, E, \epsilon, n :: \widetilde{n}) &= \mathbb{L}(S[\langle \epsilon, n\rangle], E[n \mapsto \langle \epsilon, n\rangle], \epsilon, \widetilde{n}). \end{aligned}$$

That is, $\mathbb{L}$ takes a store, an environment, a target environment handle and a list of names to bind. Each recursive call is passed a store and an environment that have been augmented with a binding for the first name in the list. Finally, the environment containing all the new bindings is added to the store. In the invocation of $\mathbb{L}$ in $C_{callfunc}$, the environment parameter $S(\epsilon_r)$ is a copy of the closure's resolution environment, which is extended by the new bindings and added to the store. $C_{callfunc}$ also pushes a $\langle \mathsf{CallFunc2}: \widetilde{n}_p, \widetilde{e}, s, \epsilon_t, \epsilon_g\rangle$ continuation, which is responsible for evaluating the actual parameters and, ultimately, scheduling the function body.

3. $C_{callfunc2}$, in conjunction with $C_{parambind}$, evaluates the actual parameters and assigns them

$$R_{returnL}: \quad \langle\langle \mathbf{Return}: e\rangle, \epsilon_L, \epsilon_G, L :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathbf{Return}: e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

$$R_{returnW}: \quad \langle\langle \mathbf{Return}: e\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e_l, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathbf{Return}: e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

$$R_{returnN}: \quad \langle\langle \mathbf{Return}: e\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{Invocation}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

$$C_{restoreenv}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{RestoreEnv}: \epsilon_l, \epsilon_g\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle u, \epsilon_l, \epsilon_g, \widetilde{K}, S\rangle$$

$$C_{invocation}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Invocation}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

Figure 3.15: Transition rules for returning from functions

to the formal parameters in the new environment. $C_{parambind}$ need not consider whether the name is already bound, since parameters take precedence over local bindings.

4. If all the formal parameters have been assigned a value, and no actual parameters remain, $C_{callfunc2nil}$ schedules the function body. The global environment of the machine is switched to the global environment that was in place when the function was defined. The local environment of the machine is switched to the new one created in the previous steps, and the old local and global environments are remembered in the $\langle \mathsf{RestoreEnv}: \epsilon_L, \epsilon_G\rangle$ continuation. The $\langle \mathsf{Invocation}\rangle$ continuation marks the point of invocation and is used to implement the $\langle \mathbf{Return}: e\rangle$ statement.

3.4.6 Returning from functions

Figure 3.15 gives the transition rules related to returning from function invocations. Since $\langle \mathbf{Return}: e\rangle$ represents an abrupt termination of the function body, it is reduced by first discarding continuations ($R_{returnL}$ and $R_{returnW}$) until an $\langle \mathsf{Invocation}\rangle$ continuation is reached, at which point the $\langle \mathsf{Invocation}\rangle$ is discarded and the expression to be returned is scheduled ($R_{returnN}$). The return value is left in the C register, which $C_{restoreenv}$ preserves, so that the calling expression's continuation can be applied to it. If a function terminates without executing a $\langle \mathbf{Return}: e\rangle$ statement, $C_{invocation}$ ensures that the return value is $\langle \texttt{None}\rangle$, which agrees with Python's behaviour.

3.4.7 Modules

The importing of modules bears some similarity to function definition and invocation. As in function definition, the result of the import is bound to a name in the local namespace, but the body of the module may be executed at import-time, and in this importing resembles function invocation. The transition rules are given in Figure 3.16; the R_{import} and $C_{importnil}$ rules are trivial, but the following are points of clarification concerning C_{import}:

$$R_{import}: \quad \langle\langle \mathbf{Import}: \widetilde{\langle n, n\rangle}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \langle \textsf{Import}: \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}, S\rangle$$

$$C_{import}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \textsf{Import}: \langle n_a, n_l\rangle :: \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}, S\rangle \longrightarrow_{MT,\nabla}$$

$$\begin{cases} \langle MT(n_a), \epsilon_1, \epsilon_1, \widetilde{K}_1, S_1\rangle & \text{if } n_a \notin \mathit{dom}(E_\nabla) \\ \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}_2, S\rangle & \text{if } S(E_\nabla(n_a)) = \langle \texttt{module} : \epsilon_2\rangle \end{cases}$$

where:

$\epsilon_1 \notin \mathit{dom}(S)$,

$E_\nabla = S(\nabla)$,

$\widetilde{K}_1 = \langle \textsf{RestoreEnv}: \epsilon_L, \epsilon_G\rangle :: \langle \textsf{Import2}: n_l, \epsilon_1\rangle :: \langle \textsf{Import}: \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}$,

$\widetilde{K}_2 = \langle \textsf{Import2}: n_l, \epsilon_2\rangle :: \langle \textsf{Import}: \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}$,

$S_1 = S[\epsilon_1 \mapsto \diamond][\nabla, n_a, \langle \texttt{module} : \epsilon_1\rangle]$

$$R_{module}: \quad \langle\langle \mathbf{Module}: s\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle s, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

$$C_{import2}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \textsf{Import2}: n, \epsilon\rangle :: \widetilde{K}, S\rangle \longrightarrow$$
$$\langle\langle \texttt{module} : \epsilon\rangle, \epsilon_L, \epsilon_G, \langle \textsf{Assign}: [\langle \mathbf{AssName}: n\rangle]\rangle :: \widetilde{K}, S\rangle$$

$$C_{importnil}: \quad \langle u, \epsilon_L, \epsilon_G, \langle \textsf{Import}: nil\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

Figure 3.16: Transition rules for importing (part 1)

- It is parameterized; it depends on a function, MT : IDENTIFIER $\rightarrow$ MODULE, and an environment handle, ∇. The MT function maps the *absolute* names of modules to their respective modules and is analogous to the part of the Python interpreter that searches the disk for the file that contains the named module. The ∇ handle refers to an environment that was reserved, when the machine was initialized, to hold the list of initialized modules. That is, in the environment $E_\nabla = S(\nabla)$, all previously initialized modules are bound by their absolute name to a $\langle \texttt{module} : \epsilon \rangle$ value.

- In the pair $\langle n_a, n_l \rangle$, n_a represents the absolute name of the module, while n_l is the name to which it should be assigned in the importing namespace (i.e. the "local" name). These will be identical unless the source code uses the following idiom:

  ```
  import absolutename as localname
  ```

  In this case, the module is bound, in the importing environment, to `localname` and not to `absolutename`.

- The rule is divided into two cases, based on whether or not the module has already been initialized, since a module is initialized at most once, no matter how many times it is imported. In the first case, where the module is to be initialized, its absolute name is bound immediately in the environment of initialized modules (i.e. in the definition of S_1), in order to curtail circular imports. An empty environment, referenced by the fresh environment handle ϵ_1, is created to serve as the new module's global environment, and the module, $MT(n_a)$, is scheduled. The R_{module} rule trivially reduces the module, $\langle \mathbf{Module} : s \rangle$, returned by MT to the module body, s, so that initialization can take place.

- Both cases push an $\langle \mathsf{Import2} : n, \epsilon \rangle$ continuation; its purpose is to schedule an assignment of the module value, $\langle \texttt{module} : \epsilon \rangle$, to the local name, n. This is not strictly necessary for the second case, where the assignment could be scheduled immediately, but the use of the same continuation in both cases is clearer. Once the module has been initialized, $C_{import2}$ schedules the assignment of the module object to the local name, which may be explicitly global.

The transition rules for the alternative form of importing, $\langle \mathbf{From} : n, \widetilde{\langle n, n \rangle} \rangle$, are given in Figure 3.17. R_{from} is similar to C_{import}, except that it does not bind the module in the local environment, and it pushes the $\langle \mathsf{From} : \epsilon, \widetilde{\langle n, n \rangle} \rangle$ continuation so that the importing of the individual names occurs after the module has been initialized. The environment handle included in this continuation refers to the environment of the imported module, so in C_{from} it is called ϵ_s, where the *s* stands for "source." The C_{from} rule takes a pair $\langle n_s, n_t \rangle$ representing a source name and a target name, finds the value of the source name, according to the source environment, and assigns the value to the target name. Finally, when all the names have been imported, $C_{fromnil}$ pops the continuation.

3.4.8 Loops

The components of the $\langle \mathbf{While} : e, s_b, s_e \rangle$ instruction are, respectively, an expression to be tested (e), a body statement to be run when the expression is true (s_b), and an else statement to be run if

R_{from} : $\langle\langle \mathbf{From}\colon n, \widetilde{\langle n, n\rangle}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow_{MT,\nabla}$

$$\begin{cases} \langle MT(n), \epsilon_1, \epsilon_1, \widetilde{K}_1, S'\rangle & \text{if } n \notin \mathit{dom}(E_\nabla) \\ \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}_2, S\rangle & \text{if } S(E_\nabla(n)) = \langle \texttt{module} : \epsilon_2\rangle \end{cases}$$

where:

$\epsilon_1 \notin \mathit{dom}(S)$,

$E_\nabla = S(\nabla)$,

$\widetilde{K}_1 = \langle \mathsf{RestoreEnv}\colon \epsilon_L, \epsilon_G\rangle :: \langle \mathsf{From}\colon \epsilon_1, \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}$,

$\widetilde{K}_2 = \langle \mathsf{From}\colon \epsilon_2, \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}$,

$S' = S[\epsilon_1 \mapsto \diamond][\nabla, n, \langle \texttt{module} : \epsilon_1\rangle]$

C_{from} : $\langle u, \epsilon_L, \epsilon_G, \langle \mathsf{From}\colon \epsilon_s, \langle n_s, n_t\rangle :: \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}, S\rangle \longrightarrow$

$\langle S(S(\epsilon_s)(n_s)), \epsilon_L, \epsilon_G, \langle \mathsf{Assign}\colon [\langle \mathbf{AssName}\colon n_t\rangle]\rangle :: \langle \mathsf{From}\colon \epsilon_s, \widetilde{\langle n, n\rangle}\rangle :: \widetilde{K}, S\rangle$

C_{fromnil} : $\langle u, \epsilon_L, \epsilon_G, \langle \mathsf{From}\colon \epsilon_s, \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$

Figure 3.17: Transition rules for importing (part 2)

$$
\begin{aligned}
R_{\mathit{while}}: &\quad \langle\langle \mathbf{While}: e, s_b, s_e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{WhileTest}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{whiletest0}}: &\quad \langle \mathit{ff}, \epsilon_L, \epsilon_G, \langle \mathsf{WhileTest}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle s_e, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
C_{\mathit{whiletest1}}: &\quad \langle \mathit{tt}, \epsilon_L, \epsilon_G, \langle \mathsf{WhileTest}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \\
&\qquad \langle s_b, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{whilerun}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \\
&\qquad \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{WhileTest}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_{\mathit{break0}}: &\quad \langle\langle \mathbf{Break}\rangle, \epsilon_L, \epsilon_G, L :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathbf{Break}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
R_{\mathit{break1}}: &\quad \langle\langle \mathbf{Break}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_{\mathit{continue0}}: &\quad \langle\langle \mathbf{Continue}\rangle, \epsilon_L, \epsilon_G, L :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathbf{Continue}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
R_{\mathit{continue1}}: &\quad \langle\langle \mathbf{Continue}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \\
&\qquad \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{WhileRun}: e, s_b, s_e\rangle :: \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.18: Transition rules for loops

the expression is false (s_e). The else statement is executed at most once (exactly once for loops that terminate normally), but the body may be executed many times, as long as the test expression is true. The transition rules for loops are given in Figure 3.18. When a while loop is initially reduced (R_{while}), all three components are remembered in the WhileTest continuation, and test expression is scheduled. The $C_{\mathit{whiletest0}}$ and $C_{\mathit{whiletest1}}$ rules take the result of the test expression and determine whether to run the loop body or the else clause. If the loop body is run, the components of the while loop remain at the top of the continuation stack, this time in a WhileRun continuation. In this way, C_{whilerun} applies after the loop body terminates normally, in order to schedule the test expression again. In the event of the abrupt termination of the loop by a **break** statement, continuations are discarded (R_{break0}) until the continuation of the loop itself is reached, and it is discarded as well (R_{break1}). For the **continue** statement, all the continuations related to the loop body are discarded ($R_{\mathit{continue0}}$), but the loop's own continuation is preserved ($R_{\mathit{continue1}}$), which effectively simulates a normal termination.

3.4.9 Conditional statements

The **if** statement is implemented by the rules in Figure 3.19. Recall that an **if** statement is represented in the abstract syntax by a list of expression-statement pairs, called tests, and an else statement, which is run if none of the test expressions succeed. Therefore, R_{if} and C_{iff1} take the first of the remaining tests, schedule the expression, and remember the statement, the remaining

$$
\begin{aligned}
R_{\mathit{if}}: &\quad \langle\langle \mathbf{If}: \langle e, s\rangle, \widetilde{\langle e, s\rangle}, s_e\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{If}: s, \widetilde{\langle e, s\rangle}, s_e\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{iff1}}: &\quad \langle \mathit{ff}, \epsilon_L, \epsilon_G, \langle \mathsf{If}: s_T, \langle e, s\rangle :: \widetilde{\langle e, s\rangle}, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{If}: s, \widetilde{\langle e, s\rangle}, s_e\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{iff2}}: &\quad \langle \mathit{ff}, \epsilon_L, \epsilon_G, \langle \mathsf{If}: s_T, \mathit{nil}, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle s_e, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \\
C_{\mathit{ift}}: &\quad \langle \mathit{tt}, \epsilon_L, \epsilon_G, \langle \mathsf{If}: s_T, \widetilde{\langle e, s\rangle}, s_e\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle s_T, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.19: Transition rules for conditional statements

$$
\begin{aligned}
R_{\mathit{print}}: &\quad \langle\langle \mathbf{Print}: \widetilde{e}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{Print}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{print}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Print}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Print}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{printnil}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Print}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_{\mathit{printnl}}: &\quad \langle\langle \mathbf{Printnl}: \widetilde{e}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \langle \mathsf{Printnl}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{printnl}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Printnl}: e :: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle e, \epsilon_L, \epsilon_G, \langle \mathsf{Printnl}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle \\
C_{\mathit{printnlnil}}: &\quad \langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Printnl}: \mathit{nil}\rangle :: \widetilde{K}, S\rangle \longrightarrow \langle\langle \mathtt{None}\rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle
\end{aligned}
$$

Figure 3.20: Transition rules for printing

tests and the else statement. If none of the test expressions succeeds, C_{iff2} schedules the else statement and pops the continuation; if a test expression does succeed, C_{ift} schedules the associated statement and pops the continuation.

3.4.10 Printing

Printing is performed by the transition rules in Figure 3.20. Since a $\langle \mathbf{Print}: \widetilde{e}\rangle$ statement contains a possibly empty list of expressions to be printed, R_{print} remembers the complete list in a $\langle \mathsf{Print}: \widetilde{e}\rangle$ continuation and places $\langle \mathtt{None}\rangle$ in the C register; the scheduling of the printed expressions is performed by C_{print}. In this way, it is possible to determine the output of an IntegerPython program by inspecting its trace for states of the form $\langle u, \epsilon_L, \epsilon_G, \langle \mathsf{Print}: \widetilde{e}\rangle :: \widetilde{K}, S\rangle$, provided the spurious occurrences of $\langle \mathtt{None}\rangle$, resulting from R_{print}, are taken into account. When no expressions remain, C_{printnil} pops the continuation. The reduction of **print** statements with line breaks ($\langle \mathbf{Printnl}: \widetilde{e}\rangle$) is performed similarly by R_{printnl}, C_{printnl} and $C_{\mathit{printnlnil}}$.

3.4.11 Miscellanea

All that remains is to provide rules for the **pass** and **global** statements. Since neither of these statements does anything when it is executed, they are reduced trivially by the rules in Figure 3.21.

$$R_{pass}: \quad \langle\langle \mathbf{Pass} \rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None} \rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$
$$R_{global}: \quad \langle\langle \mathbf{Global}: \widetilde{n} \rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle \longrightarrow \langle\langle \texttt{None} \rangle, \epsilon_L, \epsilon_G, \widetilde{K}, S\rangle$$

Figure 3.21: Transition rules for miscellaneous statements

Chapter 4

Mechanized Proof with Isabelle/HOL

In this chapter, the operational semantics from the previous chapter is embedded in the Isabelle/HOL mechanized logic and its utility for proving various kinds of theorems is discussed. The first section outlines a deep embedding of the semantics that is completed in Appendix B. The second section gives strategies for the verification of IntegerPython programs. The third section proves an invariant of the IntegerPython CEKS machine.

4.1 Embedding IntegerPython in Isabelle/HOL

In general, an embedding of a programming language within a mechanized logic can be shallow or deep. A *shallow embedding* is a direct translation from the programming language to the logical language: programs become logical formulae that are reduced according to the logic's built-in rules. A shallow embedding is, therefore, a semantics in itself, since it asserts a semantic equivalence between programming language constructs and their corresponding logical constructs. On the other hand, a *deep embedding* maintains a level of abstraction between the programming language and the logical language: programs are translated into instances of an abstract datatype, and reduction occurs by the application of a function that mimics a written semantics of the programming language. It is natural, then, given the written semantics of the previous chapter, to embed IntegerPython deeply in Isabelle/HOL, and the remainder of this section describes such an embedding. The discussion is divided into subsections for each of the following topics: abstract syntax (4.1.1), semantic values (4.1.2), environments (4.1.3), stores (4.1.4), CEKS machine states (4.1.5) and transition rules (4.1.6). The remainder of the embedding is listed in Appendix B.

4.1.1 Abstract syntax

Recall from Section 3.2 that the abstract syntax of IntegerPython consists of operators (υ, β, κ), expressions (e), targets (t) and statements (s). These are defined in Isabelle using the *datatype* command. In the case of operators, the names of the type constructors end in "Op", as in the following definition of comparison operators:

datatype *cmpop* — κ, page 23

$= \mathit{LessThanOp} \mid \mathit{LessEqOp} \mid \mathit{EqOp} \mid \mathit{NotEqOp} \mid \mathit{GreaterEqOp} \mid \mathit{GreaterThanOp}$

Unary operators and binary operators are defined similarly in Appendix B.

In the case of expressions, targets, statements and modules, the name of each type constructor is the name of the corresponding abstract syntax item, prefixed by a single letter ("e" for expressions, "t" for targets, "s" for statements, "m" for modules), in order to avoid naming conflicts with Isabelle's library. This is illustrated by the following definition of expressions:

datatype *pythonexp* — e, page 22
$= \mathit{eIntLit\ int}$
$\mid \mathit{eName\ string}$
$\mid \mathit{eGetattr\ pythonexp\ string}$
$\mid \mathit{eUop\ uop\ pythonexp}$
$\mid \mathit{eBop\ bop\ pythonexp\ pythonexp}$
$\mid \mathit{eAnd\ pythonexp\ pythonexp\ list}$
$\mid \mathit{eOr\ pythonexp\ pythonexp\ list}$
$\mid \mathit{eCompare\ pythonexp\ cmpop{*}pythonexp\ (cmpop{*}pythonexp)\ list}$
$\mid \mathit{eLambda\ string\ list\ pythonexp}$
$\mid \mathit{eCallFunc\ pythonexp\ pythonexp\ list}$

Targets, statements and modules are defined similarly in Appendix B.

4.1.2 Semantic values

The semantic values of the IntegerPython CEKS machine, as introduced in Section 3.3, consist of booleans, integers, modules, closures and $\langle \texttt{None} \rangle$. In order to define semantic values in Isabelle, we first introduce a type for environment handles, since these are contained in modules and closures. Since the *nat* type is countably infinite and totally ordered, it is sufficient for this purpose, and we define *envhandle* as a synonym of *nat*:

types
$\mathit{envhandle} = \mathit{nat}$

Semantic values may then be defined as follows:

datatype *semvalue* — u
$= \mathit{uNone}$
$\mid \mathit{uBoolean\ bool}$
$\mid \mathit{uInteger\ int}$
$\mid \mathit{uModule\ envhandle}$
$\mid \mathit{uClosure}$
pythonstmt — The body.
string list — The parameters.
string list — The bound names.
envhandle — The bindings of free names.
envhandle — The global environment at definition-time.

4.1.3 Environments

In order to define environments, we first introduce the following type, which represents locations as *envhandle-string* pairs:

datatype *location* — l
= *Location envhandle string*

Environments are then implemented using the same inductive approach with which they are represented on page 27:

datatype *environment* — E
= *EmptyEnv*
| *ExtendEnv environment string location*

In defining a function for applying an environment to a name, there is a choice between making the function partial, according to the definition of environment application on page 9, or total. Isabelle does admit partial functions, so this approach is possible, but in some types of proofs, partial functions can be cumbersome. Instead, we make use of Isabelle's *option* type, in order to define the following total function for environment application:

fun *ApplyEnv* :: *environment* ⇒ *string* ⇀ *location*
where
ApplyEnv EmptyEnv n = *None*
| *ApplyEnv* (*ExtendEnv E n′ l*) *n* =
(*if n*=*n′ then Some l else ApplyEnv E n*)

The *ApplyEnv* function returns *None* if the name is not bound by the environment, or *Some l* if the environment binds the name to the location l. In this way, the transition rules that have different cases depending on whether a name exists in an environment, such as R_{name}, may be implemented using a construct such as (*case ApplyEnv E n of None* ⇒ ... | *Some l* ⇒ ...).

Finally, the projection operator, ⊘, from page 27, is implemented by recursion on the list of projected names:

fun *ProjectEnv* :: *environment* ⇒ *string list* ⇒ *environment*
where
ProjectEnv E (*n* # *nlist*) = (*case ApplyEnv E n of*
Some l ⇒ *ExtendEnv* (*ProjectEnv E nlist*) *n l*
|*None* ⇒ *ProjectEnv E nlist*)
| *ProjectEnv E* [] = *EmptyEnv*

4.1.4 Stores

The implementation of stores is similar to that of environments, but it takes into account the three different ways of extending a store: environment handles, locations and uninitialized locations (i.e. the *DelStore* case, named for its use in the rules for deletion). The datatype definition follows:

datatype *store* — S
= *EmptyStore*

| *ExtendStoreEnv store envhandle environment*
| *ExtendStoreLoc store location semvalue*
| *DelStore store location*

Dereferencing in a store is implemented with two different functions, so that the type of each may be specified succintly. The definitions are deferred to Appendix B, but the names and types of the functions are as follows:

- *ApplyStoreLoc* :: *store* $\Rightarrow$ *location* $\rightharpoonup$ *semvalue*
- *ApplyStoreEnv* :: *store* $\Rightarrow$ *nat* $\rightharpoonup$ *environment*

The return type of *ApplyStoreLoc* is *semvalue option*, because invalid location dereferences are a legitimate run-time error that is captured by returning *None*, rather than getting stuck in the evalution of *ApplyStoreLoc*. On the other hand, *ApplyStoreEnv* dereferences an environment handle, so it ought not to fail in a consistent machine. However, for the sake of reasoning about machines that may not be consistent, and expressing what consistency means, *ApplyStoreEnv* is defined with a return type of *environment option*.

For the transition rules that require an environment handle $\epsilon \notin dom(S)$, such as $C_{callfunc}$, the following function finds a "fresh" handle by finding the greatest handle already in use and incrementing it:

function *FreshEnvHandle* :: *store* $\Rightarrow$ *envhandle*
  — finds $\epsilon \notin dom(S)$
where
  FreshEnvHandle EmptyStore = 0
| *FreshEnvHandle* (*ExtendStoreLoc S - -*) = *FreshEnvHandle S*
| *FreshEnvHandle* (*DelStore S -*) = *FreshEnvHandle S*
| *FreshEnvHandle* (*ExtendStoreEnv S* ε *-*) =
    (*if* ε >= *FreshEnvHandle S*
    then Suc ε
    else FreshEnvHandle S)
by *pat-completeness auto*
termination by *lexicographic-order*

Finally, we prove that *FreshEnvHandle* is correct by showing that the handle it returns for any given store is not in the domain of that store, where domain membership is decided by the *ExistsStoreEnv* function from Appendix B. We first prove the following lemma, which establishes a strict upper bound on the handles that can occur in the domain of a store:

lemma *fresh-envhandle-less*:
  ExistsStoreEnv S $\varepsilon \Longrightarrow \varepsilon <$ *FreshEnvHandle S*
proof (*induct S*)
  case (*ExtendStoreEnv S'* ε' *-*)
  thus *?case*
  by (*cases* $\varepsilon' < \varepsilon$) *auto*
qed *auto*

It follows that *ExistsStoreEnv S* (*FreshEnvHandle S*) $\Longrightarrow$ *FreshEnvHandle S* $<$ *FreshEnvHandle S*, which is a contradiction, so the correctness theorem follows from the contrapositive of the above lemma:

theorem *fresh-envhandle-works*:
  $\sim$ *ExistsStoreEnv S* (*FreshEnvHandle S*)
using *fresh-envhandle-less contrapos-nn*
  — *contrapos-nn*: $[\![\neg Q; P \Longrightarrow Q]\!] \Longrightarrow \neg P$
by *auto*

4.1.5 CEKS machine states

The datatype for machine states has four cases, based on the four different types that may appear in the C register: semantic values, expressions, statements and modules. This avoids defining another datatype for the C register, which would be less concise in most circumstances. The definition is as follows:

datatype *CEKS* — M
  = *uState*
    semvalue
    envhandle envhandle
    continuation list
    store
  | *eState*
    pythonexp
    envhandle envhandle
    continuation list
    store
  | *sState*
    pythonstmt
    envhandle envhandle
    continuation list
    store
  | *mState*
    pythonmodule
    envhandle envhandle
    continuation list
    store

where *continuation*, the type of continuations, is defined in Appendix B.

Finally, in a reasonable initial state for the reduction of a statement, the statement should be in C, the local and global environment registers should refer to the same (empty) environment in the store, and K should contain an empty list. The store should contain another empty environment for the module table; its handle is supplied as the ∇ parameter to the importing rules. The following constant generates such states, where the handle of the global environment is 1 and the handle of the module table is 0:

```
constdefs
  InitSState :: pythonstmt ⇒ CEKS
  [simp]: InitSState s == sState s 1 1 []
    (ExtendStoreEnv
      (ExtendStoreEnv
        EmptyStore
        0
        EmptyEnv)
      1
      EmptyEnv)
```

4.1.6 Transition rules

Most of the implementation of the transition rules is deferred to Appendix B, including the following functions:

- *ExecRuleMT* :: $(string \rightharpoonup pythonmodule) \Rightarrow envhandle \Rightarrow CEKS \Rightarrow CEKS$
- *ExecRule* :: $CEKS \Rightarrow CEKS$

The *ExecRuleMT* function takes a function mapping module names to modules, an environment handle for the list of initialized modules and a machine state; it returns the next state of the machine, as defined by the transition rules. *ExecRule* is a shorthand for *ExecRuleMT* where the function mapping module names to module bodies is empty; this is appropriate for any program that does not involve importing. These functions, which take the machine through one step, are the basis for the multi-step functions that are described in the next section.

4.2 Program Verification

A particular application of formal semantics is *program verification*, which is the process of rigorously proving that a program satisfies a given formal specification. Given an operational semantics for the language in which the program is written, one approach to program verification is to make assumptions about the initial state of the abstract machine, take the machine through enough steps to fully execute the program and prove a property of the final machine state. For the purpose of taking the IntegerPython machine through multiple steps, we define functions called *interpreters*, which fall into three categories: clocked, recursive and composed. A *clocked interpreter* is a function that executes a fixed number (called the *clock*) of steps, whereas a *recursive interpreter* is a function that continues to execute the program until a more general termination condition is met. (For a discussion of the relative merits of clocked and recursive interpreters, see [17].) Moreover, a *composed interpreter* is constructed from other interpreters by function composition.

The clocked interpreter for IntegerPython is defined as follows:

```
fun CycleAllRulesMT ::
```

```
    (string ⇀ pythonmodule) ⇒ envhandle ⇒ nat ⇒ CEKS ⇒ CEKS
where
    CycleAllRulesMT MT vN 0 s = s
|   CycleAllRulesMT MT vN (Suc n) s =
        ExecRuleMT MT vN (CycleAllRulesMT MT vN n s)

constdefs
    CycleAllRules :: nat ⇒ CEKS ⇒ CEKS
    [simp]: CycleAllRules == CycleAllRulesMT (λx. None) arbitrary
```

Numerical constants (e.g. *8*, *11*) are not useful as clock arguments, since Isabelle does not match them with the *Suc n* pattern. Therefore, we define the following constants for expressing clocks:

```
constdefs
    One :: nat
    [simp]: One == Suc 0
    Two :: nat
    [simp]: Two == Suc One
    Three :: nat
    [simp]: Three == Suc Two
    Four :: nat
    [simp]: Four == Suc Three
    Five :: nat
    [simp]: Five == Suc Four
    Six :: nat
    [simp]: Six == Suc Five
    Seven :: nat
    [simp]: Seven == Suc Six
    Eight :: nat
    [simp]: Eight == Suc Seven
    Nine :: nat
    [simp]: Nine == Suc Eight
    Ten :: nat
    [simp]: Ten == Suc Nine
    Twenty :: nat
    [simp]: Twenty == Ten + Ten
    Thirty :: nat
    [simp]: Thirty == Twenty + Ten
```

A clock of 11 can thus be written as *Ten* + *One* or *Suc Ten*.

The recursive interpreter is called *CycleUntil*, because it recurs until the machine state satisfies a given predicate. Since this function does not terminate for all inputs (e.g. suppose the predicate were λ*x*. *False*), the termination proof is omitted from its definition, which follows:

```
function (tailrec)
    CycleUntil :: (CEKS ⇒ bool) ⇒ CEKS ⇒ CEKS
where
```

until-simp [*simp del*]:
CycleUntil f s = (*if* (*f s*) *then s else CycleUntil f* (*ExecRule s*))
by *auto*

The definitional axiom for *CycleUntil* is also omitted from the simpset, because it can easily cause the simplifier to loop. Indeed, looping occurs if the machine never reaches a terminal state, but it also occurs when the termination predicate, *f s*, cannot be simplified. Therefore, we introduce a safer simplification rule for *CycleUntil* as follows:

lemma *sane-until-simp*:
¬ *f s* ⟹ *CycleUntil f s* = *CycleUntil f* (*ExecRule s*)
by (*simp add*: *until-simp*)

Because it has ¬ *f s* as a premise, it does not apply unless the termination predicate can be simplified, and so it prevents some causes of looping in the simplifier. It is therefore useful for interactive proof development and for certain kinds of finished proofs.

Composed interpreters arise where a proof about a program incorporates existing proofs about its constituent code fragments. For example, if a code block invokes a function, and a lemma has been proven concerning that function's effect on the machine state, the lemma could be incorporated into the proof about the calling code. However, the proof must be structured in such a way as to make the lemma applicable, and there are a number of obstacles to this:

- If the lemma uses a recursive interpreter, it is not useful in a result about a clocked interpreter, and vice versa.
- If both use a recursive interpreter, the termination predicate is likely to be different, which prevents the simplifier from recognizing that the lemma is related to the main result.
- If both use a clocked interpreter, arithmetic simplifications of the combined clock may prevent the simplifier from using the lemma.

An approach for combining clocks using non-arithmetic operators is described in [7], and an approach for "composing" clock-based proofs and "inductive invariant" proofs is described in [28]. However, it seems natural to achieve the composition of proofs by the straight-forward composition of interpreters. For example, consider the following IntegerPython fragment:

```
a = 1
func()
b = 5
```

It takes 11 steps (R_{stmt}, R_{assign}, $R_{intconst}$, $C_{nassign}$, $C_{assignnil}$, C_{stmt}, $R_{discard}$, $R_{callfunc}$, R_{name}, $C_{callfunc}$, $C_{callfunc2nil}$) to execute `a = 1` and invoke `func`. After `func` returns, it takes eight steps ($C_{restoreenv}$, $C_{discard}$, C_{stmt}, R_{assign}, $R_{intconst}$, $C_{nassign}$, $C_{assignnil}$, $C_{stmtnil}$) to finish the execution. Supposing there is an interpreter, *FuncInterpreter*, for the body of `func`, the following is an interpreter for the calling code fragment: $\lambda s.$ *CycleAllRules Eight* (*FuncInterpreter* (*CycleAllRules* (*Suc Ten*) *s*)). Once the innermost interpreter has been simplified to a state, the application of *FuncInterpreter* may be rewritten by an applicable lemma, although it may be necessary to remove the definition of

FuncInterpreter from the simpset to prevent the simplifier from using that definition instead of the lemma. In this way, program verification with our mechanized semantics typically involves isolating sections of the program to be treated seperately, defining interpreters for these sections, proving their properties and composing an interpreter for the aggregate program from these proven sub-interpreters.

4.3 Invariants

An *invariant* is a property of an abstract machine that is preserved by all its legal transitions. That is, if an invariant holds for a given state, it holds for that state's successor in the transition relation. In particular, if an invariant holds for the initial state of a machine, it holds for every state the machine reaches. Thus, invariants are a useful tool for studying the consistency of a language and its semantics.

As an example of an invariant, consider the property of IntegerPython environments introduced in Section 3.3: $E(n) = \langle \epsilon', n' \rangle$ implies $n = n'$. For a single environment, this property is phrased in Isabelle/HOL as follows:

fun *TheSameName* :: *environment* ⇒ *bool*
where
  TheSameName (*ExtendEnv E n l*) = (*case l of Location - n'* ⇒ ((*n* = *n'*) & *TheSameName E*))
| *TheSameName EmptyEnv* = *True*

That the property holds for every environment in a machine is expressed by *SNInvariant* and its helper functions:

constdefs
  AllStates :: (*envhandle* ⇒ *envhandle* ⇒ (*continuation list*) ⇒ *store* ⇒ *'a*) ⇒ *CEKS* ⇒ *'a*
  [*simp*]: (*AllStates f s*) ≡ (*case s of*
    uState u vL vG K S ⇒ (*f vL vG K S*)
    |*eState e vL vG K S* ⇒ (*f vL vG K S*)
    |*sState s vL vG K S* ⇒ (*f vL vG K S*)
    |*mState m vL vG K S* ⇒ (*f vL vG K S*))

fun *SameName* :: *environment option* ⇒ *bool*
where
  SameName None = *True*
| *SameName* (*Some E*) = *TheSameName E*

constdefs
  SNInvariant :: *CEKS* ⇒ *bool*
  [*simp*]: *SNInvariant s* ≡ (*AllStates*
    (λ *vL vG K S*. (∀ *v*. *SameName* (*ApplyStoreEnv S v*))) *s*)

The goal, then, is to show that, for any state *s*, *SNInvariant s* ⟹ *SNInvariant* (*ExecRule s*). The proof is conducted by splitting the cases of *s* until it is clear which transition rule applies to each case. The proof is trivial for transition rules that do not change any environments, but

other rules require subproofs. Especially challenging are the rules that change an environment *en masse*, either by creating a list of local variables, or projecting out a list of free variables. The following lemmas show that the $\mathbb{L}$ function on page 38, as implemented by *CreateLocals* and *CreateLocalsRec* in Appendix B, preserves the invariant:

declare *Let-def* [*simp*]

lemma *createlocalsrec-theenv* [*simp*]:
ApplyStoreEnv (*fst* (*CreateLocalsRec* (*S*, *E*) *v B*)) *v'* = *ApplyStoreEnv S v'*
proof (*induct B*)
  case (*Cons - b2n*)
  thus *?case*
  by (*cases CreateLocalsRec* (*S*, *E*) *v b2n*) *simp*
qed *simp*

lemma *createlocals-theenv-ne* [*simp*]:
assumes *v* ~= *v'*
shows *ApplyStoreEnv* (*CreateLocals S E B v'*) *v* = *ApplyStoreEnv S v*
using *assms*
proof (*cases B*)
  case *Cons*
  have *one*: *CreateLocals S E B v'* =
    ExtendStoreEnv
      (*fst* (*CreateLocalsRec* (*S*, *E*) *v' B*))
      v' (*snd* (*CreateLocalsRec* (*S*, *E*) *v' B*))
    using *prems* **by** (*case-tac CreateLocalsRec* (*S*, *E*) *v' B*) *simp*
  with *assms* **have** *two*: *ApplyStoreEnv* (*CreateLocals S E B v'*) *v* =
    ApplyStoreEnv (*fst* (*CreateLocalsRec* (*S*, *E*) *v' B*)) *v*
    by *simp*
  thus *?thesis* **by** *simp*
qed *simp*

lemma *createlocals-theenv-eq* [*simp*]:
assumes *v* = *v'*
shows *ApplyStoreEnv* (*CreateLocals S E B v'*) *v* = *Some* (*snd* (*CreateLocalsRec* (*S*, *E*) *v' B*))
using *assms*
proof (*cases B*)
  case *Cons*
  have *one*: *CreateLocals S E B v'* =
    ExtendStoreEnv
      (*fst* (*CreateLocalsRec* (*S*, *E*) *v' B*))
      v' (*snd* (*CreateLocalsRec* (*S*, *E*) *v' B*))
    using *prems* **by** (*case-tac CreateLocalsRec* (*S*, *E*) *v' B*) *simp*
  with *assms* **show** *?thesis* **by** *simp*
qed *simp*

lemma *samename-cl*:
assumes
TheSameName E
SameName (*ApplyStoreEnv S v'*)
shows
SameName (*ApplyStoreEnv* (*CreateLocals S E B v*) *v'*)
using *assms*
by (*induct B*) (*auto split*: *prod.split*)

Moreover, projection with the ⊘ operator preserves the invariant:

lemma *project-empty* [*simp*]: *ProjectEnv EmptyEnv anylist* = *EmptyEnv*
by (*induct anylist*) *simp-all*

lemma *samename-contained*:
TheSameName E ⟹ *ApplyEnv E n* = *Some* (*Location v nl*) ⟹ *n* = *nl*
proof (*induct E*)
case (*ExtendEnv E' n' l'*)
show *?case* **using** *prems*
proof (*cases l'*)
case (*Location vl' nl'*)
show *?thesis* **using** *prems*
by (*cases n* = *nl'*) *simp-all*
qed
qed *simp*

lemma *samename-project*: *TheSameName E* ⟹ *TheSameName* (*ProjectEnv E anylist*)
proof (*induct anylist*)
case (*Cons n1 n2n*)
with *samename-contained* **show** *?case*
by (*auto split*: *location.split option.split*)
qed *simp* **declare** *Let-def* [*simp del*]

The case analysis that proves the invariant is lengthy and is deferred to Appendix C. However, because the proof is conducted in Isabelle/HOL, we may be confident that all cases have indeed been analyzed and that the property is truly invariant. It follows that the IntegerPython language is not capable of creating aliases within the same namespace.

Chapter 5

Analysis, Future Work and Conclusions

In this chapter, we analyze the IntegerPython semantics in terms of its suitability for further development. The first section analyzes the efficiency of the IntegerPython semantics and outlines strategies for the implementation of efficient IntegerPython interpreters. The second section discusses the four-stage development plan introduced in Chapter 1 and suggests implementations for the remaining features of Python. The third section concludes the report with a summary of the important results of each chapter.

5.1 Efficiency

The implementation of the IntegerPython semantics in Chapter 4 provides for symbolic execution of IntegerPython programs within the Isabelle/HOL environment, but it does not immediately provide for their execution as programs of a physical computer. Since IntegerPython is a subset of Python, the reference Python interpreter might be used to execute IntegerPython programs on a physical computer, but it may not give the same results as the CEKS machine. Thus, even programs that are proven on the CEKS machine may fail to give the expected results on the physical machine. It is useful, therefore, to construct an interpreter that follows the operational semantics exactly, provided such an interpreter can be made reasonably efficient.

In general, the operational semantics of IntegerPython, as presented in this report, is optimized for clarity of presentation and for ease of use in proof scripts. As a result, it is less than optimal in terms of time and space complexity, especially with regard to the representation of the store and the procedure for finding the value of variables. The space efficiency of the store can be improved by implementing *garbage collection*, and algorithms for this purpose are surveyed in [34]. With respect to the time efficiency of store application, the $\langle \epsilon, n \rangle$ format for locations, which is useful in proofs, is inefficient in execution, since it requires string comparisons. If the store is implemented directly in physical memory, locations need not be anything more than memory addresses.

The second source of inefficiency concerns the R_{name} rule on page 30. It contains three cases, which are distinguished at run-time by potentially many string comparisons. However, it is known at *parse-time* which of the cases will apply, so the efficiency of name lookup can be improved by including more information from the parser in the abstract syntax. Specifically, the $\langle \mathbf{Name} : n \rangle$

AST node could be replaced by three distinct nodes (e.g. $\langle \mathbf{NameNone} \rangle$, $\langle \mathbf{NameLocal} : n \rangle$, $\langle \mathbf{NameGlobal} : n \rangle$) and the cases of R_{name} split into separate rules. Moreover, it may be possible to optimize the implementation of $\langle \mathbf{NameLocal} : n \rangle$ by further dividing it into cases for names that are free, locally bound, or locally bound but not referenced in enclosed functions. In the last case, comparison of names could be completely replaced by numerical offsets, as in the stack-based allocation of local variables that is common in compiled languages. Once name lookup and the store representation have been suitably optimized, it should be possible to construct an efficient interpreter from the IntegerPython semantics.

5.2 Extensibility

Recall that IntegerPython represents only the first stage of a four-stage plan for full feature compliance, as described in Table 1.1 on page 5. In this section, we explain the reasons for the proposed division of features and outline strategies for implementing the remaining features with minimal changes to the data model of IntegerPython.

The new features of the proposed second stage, ObjectPython, are a minimal implementation of the "standard type hierarchy" ([33], §3.2), strings, dictionaries, "new-style" classes ([33], §3.3) and exceptions. Since everything in Python is an object, introducing the object model as soon as possible allows other types, such as lists and tuples, to be implemented in full generality, rather than with *ad hoc* rules that must later be restated to bring the types into the object model. It may also permit some features to be implemented with a library of ObjectPython code, rather than with new transition rules. Strings and dictionaries, which are mappings from strings to values (i.e. store locations), are introduced as fundamental types at this level because they are used in Python's object model to implement the namespaces of classes and modules ([33], §3.2). That is, environments are replaced by the more general dictionary type, and ϵ becomes a dictionary handle. An object, then, consists of a dictionary for attributes and a list of read-only attributes; any implementation data, such as subclass relationships, etc. can be stored in a read-only attribute. The revised semantic value grammar might look like the following:

$$\begin{array}{rcrcl} \textsc{Value} & \ni & u & ::= & \langle \texttt{True} \rangle \mid \langle \texttt{False} \rangle \mid i \mid \langle \texttt{None} \rangle \\ & & & \mid & \langle \texttt{string} : \text{"} \ldots \text{"} \rangle \\ & & & \mid & \langle \texttt{dict} : \epsilon \rangle \\ & & & \mid & \langle \texttt{object} : \epsilon, \widetilde{n} \rangle \end{array}$$

Modules and closures are absent, since they can be implemented as general objects. Lastly, including exceptions in the second stage limits the number of rules that need to be restated. That is, once exceptions are included, it is no longer sufficient to let the CEKS machine get stuck when there is a run-time error; any transition rule that was only partial must be made total by adding cases that raise an exception. Exception handling can be implemented with a distinguished continuation scheme similar to that already used by the **break**, **continue** and **return** statements.

The proposed third stage, StandardPython, contains all the remaining features except the "classic" object model ([33], §3.3). It completes the standard type hierarchy, and implements such features as generators, subscription and slicing in full, object-oriented generality. This stage, though

potentially large, builds on the notational core established in the first two stages, so that its development should prove mostly mechanical.

The classic object model, which was the only object model in versions of Python prior to 2.2, is left to the final stage, because it is a legacy feature that is destined to be "dropped [in favour of] new-style classes" ([33], §3.3). The lack of classic objects may be a compatibility issue for third-party Python code, but it should be possible to convert the standard library to the new-style model with little or no inaccuracy. It may also be possible to approximate classic objects in ObjectPython, because the two object models differ primarily in the semantics of inheritance and subtyping, and the new-style model is more expressive in this respect. Therefore, much work remains to be done in the development of a complete semantics for Python, but the semantics of IntegerPython constitutes a feasible starting point.

5.3 Conclusions

The stated goal of this report is "to demonstrate a formal definition of a subset of a real-world scripting language and the use of such a definition in mechanically verifiable proofs" (page 4). This has been accomplished for the IntegerPython subset of the Python language.

In Chapter 3, we developed an operational semantics of IntegerPython on a CEKS abstract machine. This semantics specified a notation for store locations that simplifies proofs about blocks of code, compared to choosing locations from an arbitrary infinite set. Moreover, the semantics allowed for randomly accessible namespaces by placing environments in the store and referencing them with environment handles. A second environment register was added to the machine to store the bindings of global variables; the resolution of free variables from enclosing namespaces was handled by static analysis, so that only one local environment was needed at any given program point. Module importing was implemented by reserving an environment to contain all previously initialized modules.

In Chapter 4, we embedded the semantics of IntegerPython in the HOL logic of the Isabelle proof assistant. This embedding defined clocked and recursive interpreters for IntegerPython, so that interpreters for an aggregate program could be composed from proven interpreters for individual parts of the program. The use of Isabelle/HOL for program verification was outlined. An invariant of the CEKS machine was proven, which established that no two IntegerPython variables in the same namespace may be aliases of each other.

Finally, in Chapter 5, we have outlined strategies for the efficient, executable implementation of the IntegerPython semantics and for the extension of the semantics toward full compliance with the Python language.

Bibliography

[1] Isabelle/HOL Theory Library. Bundled with Isabelle, http://isabelle.in.tum.de/, 2007.

[2] ACL2 Theory Library. Bundled with ACL2, http://www.cs.utexas.edu/~moore/acl2/, August 2008. Version 3.4.

[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory*. Computer Science and Applied Mathematics. Academic Press, 1986.

[4] B. E. Aydemir, A. Bohannon, J. N. F. Matthew Fairbairn, B. C. Pierce, D. V. Peter Sewell, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65, Oxford, UK, Aug. 2005. Springer.

[5] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. *sefm*, 0:200–209, 2005.

[6] R. S. Boyer and J. S. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[7] R. S. Boyer and J. S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, chapter 7, pages 147–176. MIT Press, 1997.

[8] J. Clements and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.*, 26(6):1029–1052, 2004.

[9] W. D. Clinger. Proper tail recursion and space efficiency. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, Montreal, Canada, 1998.

[10] M. Felleisen and D. P. Friedman. A reduction semantics for imperative higher-order languages. In *PARLE Parallel Architectures and Languages Europe, Volume I*, Lecture Notes in Computer Science, pages 206–223. Springer-Verlag, 1987.

[11] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 8, pages 113–135. Kluwer, 2000.

[12] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[13] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[15] H. Liu and J. S. Moore. Executable JVM model for analytical reasoning: A study. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 15–23, San Diego, California, USA, June 2003.

[16] H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. In *Proceedings of the 17th International Conference in Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200, Park City, Utah, USA, Sept. 2004. Springer.

[17] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, October 2003.

[18] J. S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design: Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 139–162. Springer, 1999.

[19] J. S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.

[20] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86^{TM} floating-point division program. *IEEE Trans. Comput.*, 47(9):913–926, 1998.

[21] J. S. Moore and G. M. Porter. An executable formal Java Virtual Machine thread model. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 91–103, Monterey, California, USA, Apr. 2001.

[22] L. Moreau. Correctness of a distributed-memory model for Scheme. In *In Second International Europar Conference (EURO-PAR'96), number 1123 in Lecture Notes in Computer Science*, pages 615–624, Lyon, France, 1996. Springer-Verlag.

[23] T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, San Diego, CA, USA, Jan. 19–21, 1998. ACM Press, New York.

[24] T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000. http://isabelle.in.tum.de/Bali/papers/MOD99.html.

[25] D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[26] L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, New York, NY, USA, 2006. ACM.

[27] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1999.

[28] S. Ray and J. S. Moore. Proof styles in operational semantics. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *Lecture Notes in Computer Science*, pages 67–81, Austin, Texas, USA, Nov. 2004. Springer.

[29] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/devel-entries/Simpl.shtml, Feb. 2008.

[30] G. L. Steele. *Common LISP. The language. 2nd ed.* Digital Press, Woburn, MA, 1990.

[31] S. Swords and W. Cook. Soundness of the simply typed lambda calculus in ACL2. In *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'06)*, Seattle, Washington, USA, Aug. 2006. Available online at http://www-static.cc.gatech.edu/~manolios/acl206/program.html.

[32] G. van Rossum. *Python Library Reference*. Python Software Foundation, release 2.5.1 edition. http://docs.python.org/ref/ref.html.

[33] G. van Rossum. *Python Reference Manual (Release 2.5.1)*. Python Software Foundation, Apr. 2007. Available online at http://www.python.org/doc/2.5.1/ref/ref.html.

[34] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, Sept. 1992.

Appendix A

Concrete Syntax for IntegerPython

This appendix gives the concrete syntax of IntegerPython.

A.1 Grammar

Extended Backus-Naur Form (EBNF) is employed to specify the concrete syntax of IntegerPython: double quotation marks (‘"’) for delimiting terminal symbols, asterisk (‘*’) for Kleene star, plus (‘+’) for Kleene plus, vertical bar (‘|’) for alternation, square brackets (‘[’ and ‘]’) for delimiting optional elements, and round brackets (‘(’ and ‘)’) for grouping. The grammar specifies a proper subset of Python: every grammatically valid IntegerPython program is also a grammatically valid Python program. The grammar is adopted from the one in the official Python language reference [33]. The lexical aspects of the syntax is omitted for brevity.

```
atom ::=
             identifier | literal | enclosure

enclosure ::=
             parenth_form

literal ::=
             integer | longinteger

parenth_form ::=
             "(" expression ")"

primary ::=
             atom | attributeref | call

attributeref ::=
             primary "." identifier
```

```
call ::=
            primary "(" [argument_list [","]] ")"

argument_list ::=
            positional_arguments

positional_arguments ::=
            expression ("," expression)*

u_expr ::=
            "-" u_expr | "+" u_expr | "~" u_expr

m_expr ::=
            u_expr
             | m_expr "*" u_expr
             | m_expr "//" u_expr
             | m_expr "/" u_expr
             | m_expr "%" u_expr

a_expr ::=
            m_expr
             | a_expr "+" m_expr
             | a_expr "-" m_expr

shift_expr ::=
            a_expr

and_expr ::=
            shift_expr

xor_expr ::=
            and_expr

or_expr ::=
            xor_expr

comparison ::=
            or_expr ( comp_operator or_expr )*

comp_operator ::=
            "<" | ">" | "==" | ">=" | "<=" | "<>" | "!="
```

```
expression ::=
             or_test [if or_test else test] | lambda_form

or_test ::=
             and_test | or_test "or" and_test

and_test ::=
             not_test | and_test "and" not_test

not_test ::=
             comparison | "not" not_test

lambda_form ::=
             "lambda" [parameter_list]: expression

simple_stmt ::=
             expression_stmt
              | assignment_stmt
              | pass_stmt
              | del_stmt
              | print_stmt
              | return_stmt
              | break_stmt
              | continue_stmt
              | import_stmt
              | global_stmt

expression_stmt ::=
             expression

assignment_stmt ::=
             (target "=")+ expression

target ::=
             identifier | attributeref

pass_stmt ::=
             "pass"

del_stmt ::=
             "del" target
```

```
print_stmt ::=
             "print" [expression ("," expression)* [","]]

return_stmt ::=
             "return" [expression]

break_stmt ::=
             "break"

continue_stmt ::=
             "continue"

import_stmt ::=
             "import" module ["as" name] ( "," module ["as" name] )*
              | "from" module "import" identifier ["as" name]
                 ( "," identifier ["as" name] )*
              | "from" module "import" "(" identifier ["as" name]
                 ( "," identifier ["as" name] )* [","] ")"

module ::=
             (identifier ".")* identifier

global_stmt ::=
             "global" identifier ("," identifier)*

compound_stmt ::=
             if_stmt
              | while_stmt
              | funcdef

suite ::=
             stmt_list NEWLINE
              | NEWLINE INDENT statement+ DEDENT

statement ::=
             stmt_list NEWLINE | compound_stmt

stmt_list ::=
             simple_stmt (";" simple_stmt)* [";"]

if_stmt ::=
```

```
              "if" expression ":" suite
               ( "elif" expression ":" suite )*
               ["else" ":" suite]

while_stmt ::=
              "while" expression ":" suite
               ["else" ":" suite]

funcdef ::=
              "def" funcname "(" [parameter_list] ")"
               ":" suite

parameter_list ::=
              (defparameter ",")* defparameter [","]

defparameter ::=
              parameter

parameter ::=
              identifier

funcname ::=
              identifier

file_input ::=
              (NEWLINE | statement)*
```

A.2 Summary of Differences with Python

The syntactic difference between IntegerPython and Python [33] is summarized as follows. Generally, any syntactic construct that involves the manipulation of data types other than integers is removed from the grammar.

- Those literal types other than integers or long integers are not supported.
- Expression lists are not supported because they produce tuples. For example, `parenth_form` now contains a mandatory `expression` rather than an optional `expression_list`. Similar restrictions have been applied to `expression_stmt`, `assignment_stmt`, `augmented_assignment_stmt` and `return_stmt`.
- In an `enclosure`, no `list_display`, `dict_display`, `generator_expression`, or `string_conversion` is allowed.
- In a `primary`, no `subscription` or `slicing` is allowed.

- In an `argument_list`, no star, double-star or keyword argument is allowed.
- In a `comp_operator`, membership tests (`in`) are not allowed.
- In a `simple_stmt`, the following statement types are not allowed: `assert_stmt`, `augmented_assi` `yield_stmt`, `raise_stmt`, `exec_stmt`.
- Pattern matching is not supported. Rather than a `target_list`, a single `target` now appears on the left of "=". In addition, a `target` can only be an `identifier` or `attributeref`.
- The print statement does not allow the specification of output destination (">>").
- Importing "$\star$" is no longer supported.
- A `compound_stmt` may no longer contain the following statement types: `for_stmt`, `try_stmt`, `with_stmt` and `classdef`.
- A `funcdef` may no longer contain `decorators`.
- The star and double-star parameters, and default values, are no longer supported in a `parameter_list`.
- The `power` case of `u_expr` is no longer supported.
- No `sub_list` may appear as a `parameter`.

Appendix B

IntegerPython Isabelle Code

B.1 Abstract Syntax

datatype *uop* — υ, page 22
= *UnaryPlusOp* | *UnaryMinusOp* | *UnaryInvertOp* | *UnaryNotOp*

datatype *bop* — β, page 23
= *AddOp* | *SubOp* | *MulOp* | *DivOp* | *ModOp*

datatype *pythontarget* — t
= *tAssAttr pythonexp string*
| *tAssName string*
| *tDelAttr pythonexp string*
| *tDelName string*

datatype *pythonstmt* — s
= *sAssign pythontarget list pythonexp*
| *sBreak*
| *sContinue*
| *sDiscard pythonexp*
| *sFrom string* (*string*∗*string*) *list*
| *sFunction string string list pythonstmt*
| *sGlobal string list*
| *sIf pythonexp* ∗ *pythonstmt* (*pythonexp*∗*pythonstmt*) *list pythonstmt*
| *sImport* (*string*∗*string*) *list*
| *sPass*
| *sReturn pythonexp*
| *sStmt pythonstmt pythonstmt list*
| *sWhile pythonexp pythonstmt pythonstmt*
| *sPrint pythonexp list*
| *sPrintnl pythonexp list*

datatype *pythonmodule* — m
= *mModule pythonstmt*

B.2 Stores

function *ApplyStoreLoc* :: *store* => *location* => *semvalue option*
— $S(l)$
where
ApplyStoreLoc EmptyStore - = None
| *ApplyStoreLoc* (*ExtendStoreEnv S - -*) *l* = (*ApplyStoreLoc S l*)
| *ApplyStoreLoc* (*ExtendStoreLoc S l' u*) *l* =
(*if l=l' then Some u else ApplyStoreLoc S l*)
| *ApplyStoreLoc* (*DelStore s l'*) *l* =
(*if l=l' then None else* (*ApplyStoreLoc s l*))
by *pat-completeness auto*
termination by *lexicographic-order*

function *ExistsStoreLoc* :: *store* => *location* => *bool*
— $l \in dom(S)$
where
ExistsStoreLoc EmptyStore - = False
| *ExistsStoreLoc* (*ExtendStoreLoc S l' -*) *l* = ($l' = l$ | *ExistsStoreLoc S l*)
| *ExistsStoreLoc* (*DelStore S l'*) *l* = ($l' = l$ | *ExistsStoreLoc S l*)
| *ExistsStoreLoc* (*ExtendStoreEnv S - -*) *l* = *ExistsStoreLoc S l*
by *pat-completeness auto*
termination by *lexicographic-order*

constdefs — $l \in init(S)$
InitStoreLoc :: *store* => *location* => *bool*
[*simp*]: *InitStoreLoc S l* == (*case ApplyStoreLoc S l of*
None => *False*
|*Some -* => *True*)

function *ApplyStoreEnv* :: *store* => *envhandle* ~=> *environment*
— $S(\epsilon)$
where
ApplyStoreEnv EmptyStore - = None
| *ApplyStoreEnv* (*ExtendStoreEnv S* ε' *E*) ε =
(*if* ε'=ε *then Some E else ApplyStoreEnv S* ε)
| *ApplyStoreEnv* (*ExtendStoreLoc S - -*) ε = *ApplyStoreEnv S* ε
| *ApplyStoreEnv* (*DelStore S -*) ε = *ApplyStoreEnv S* ε
by *pat-completeness auto*

termination by *lexicographic-order*

constdefs — $\epsilon \in dom(S)$
ExistsStoreEnv :: *store* => *envhandle* => *bool*
[*simp*]: *ExistsStoreEnv S* ε == (*case ApplyStoreEnv S* ε *of*
None => *False*
|*Some* - => *True*)

B.3 Binary Operations

constdefs
bool2int :: *bool* => *int*
[*simp*]: *bool2int b* == (*if b then 1 else 0*)
constdefs
BopEvalInt :: *bop* => *int* => *int* ~=> *semvalue*
[*simp*]: *BopEvalInt bop iL iR* == (*case bop of*
AddOp => *Some* (*uInteger* (*iL* + *iR*))
|*SubOp* => *Some* (*uInteger* (*iL* − *iR*))
|*MulOp* => *Some* (*uInteger* (*iL* ∗ *iR*))
|*DivOp* => *Some* (*uInteger* (*iL div iR*))
|*ModOp* => *Some* (*uInteger* (*iL mod iR*))
)

fun *BopEval* :: *bop* => *semvalue* => *semvalue* => *semvalue option*
where
BopEval bop (*uInteger iL*) (*uInteger iR*) =
BopEvalInt bop iL iR
| *BopEval bop* (*uInteger iL*) (*uBoolean bR*) =
BopEvalInt bop iL (*bool2int bR*)
| *BopEval* - (*uInteger* -) - = *None*
| *BopEval bop* (*uBoolean bL*) (*uInteger iR*) =
BopEvalInt bop (*bool2int bL*) *iR*
| *BopEval bop* (*uBoolean bL*) (*uBoolean bR*) =
BopEvalInt bop (*bool2int bL*) (*bool2int bR*)
| *BopEval* - (*uBoolean* -) - = *None*
| *BopEval* - - - = *None*

B.4 Unary Operations

constdefs

```
    UopEvalInt :: uop => int => semvalue option
    [simp]: UopEvalInt uop i == (case uop of
        UnaryPlusOp => Some (uInteger i)
        |UnaryMinusOp => Some (uInteger (-1 * i))
        |UnaryInvertOp => Some (uInteger (-1 * (i + 1)))
        |UnaryNotOp => Some (uBoolean (i = 0)))

function UopEval :: uop => semvalue => semvalue option
where
    UopEval uop (uInteger i) = UopEvalInt uop i
|   UopEval uop (uBoolean b) = UopEvalInt uop (bool2int b)
|   UopEval uop (uClosure - - - - -) = (case uop of
        UnaryNotOp => Some (uBoolean False)
        |- => None)
|   UopEval uop (uModule -) = (case uop of
        UnaryNotOp => Some (uBoolean False)
        |- => None)
|   UopEval uop uNone = (case uop of
        UnaryNotOp => Some (uBoolean True)
        |- => None)
by pat-completeness auto
termination by lexicographic-order
```

B.5 Comparison Operations

```
constdefs
    EvalLess :: semvalue => semvalue => bool
    [simp]: EvalLess uL uR == (case uL of
        uNone => (uR ~= uNone)
        |uClosure - - - vL - => (case uR of
            uNone => False
            |uClosure - - - vR - => vL < vR
            |- => True)
        |uModule vL => (case uR of
            uNone => False
            |uClosure - - - - - => False
            |uModule vR => vL < vR
            |- => True)
        |uInteger iL => (case uR of
            uInteger iR => (iL < iR)
            |uBoolean bR => (iL < bool2int bR)
            |- => False)
        |uBoolean bL => (case uR of
            uInteger iR => (bool2int bL < iR)
```

$|uBoolean\ bR \Rightarrow (bool2int\ bL < bool2int\ bR)$
$|\text{-} \Rightarrow False))$
$CmpopEval :: cmpop \Rightarrow semvalue \Rightarrow semvalue \Rightarrow bool$
$[simp]: CmpopEval\ \kappa\ uL\ uR == (case\ \kappa\ of$
$LessThanOp \Rightarrow (EvalLess\ uL\ uR)$
$|LessEqOp \Rightarrow (EvalLess\ uL\ uR \mid (uL = uR))$
$|EqOp \Rightarrow (uL = uR)$
$|NotEqOp \Rightarrow (uL \sim= uR)$
$|GreaterThanOp \Rightarrow (EvalLess\ uR\ uL)$
$|GreaterEqOp \Rightarrow (EvalLess\ uR\ uL \mid (uL = uR)))$

B.6 Continuations

datatype *continuation* — K
$= AssignCont\ pythontarget\ list$
$\mid AssAttrCont\ string\ semvalue$
$\mid BopLeftCont\ bop\ pythonexp$
$\mid BopRightCont\ bop\ semvalue$
$\mid CallFuncCont\ pythonexp\ list$
$\mid CallFuncTwoCont$
string list — The formal parameters
pythonexp list — The actual parameters
pythonstmt — The function body
envhandle — The invocation's local environment
envhandle — The function's global environment
$\mid CompareCont\ cmpop{*}pythonexp\ (cmpop{*}pythonexp)\ list$
$\mid CompareTwoCont\ semvalue\ cmpop\ (cmpop{*}pythonexp)\ list$
$\mid DelAttrCont\ string\ semvalue$
$\mid DiscardCont$
$\mid FromCont\ envhandle\ (string{*}string)\ list$
$\mid GetAttrCont\ string$
$\mid IfCont$
pythonstmt — The pending statement.
$(pythonexp{*}pythonstmt)\ list$ — The remaining tests
pythonstmt — The else clause.
$\mid ImportCont\ (string{*}string)\ list$
$\mid ImportTwoCont\ string\ envhandle$
$\mid InvocationCont$
$\mid ParamBindCont\ string\ envhandle$
$\mid PrintCont\ pythonexp\ list$
$\mid PrintnlCont\ pythonexp\ list$
$\mid RestoreEnvCont\ envhandle\ envhandle$
$\mid ScAndCont\ pythonexp\ list$
$\mid ScOrCont\ pythonexp\ list$
$\mid StmtCont\ pythonstmt\ list$

```
  | UopCont uop
  | WhileRunCont pythonexp pythonstmt pythonstmt
  | WhileTestCont pythonexp pythonstmt pythonstmt
```

B.7 List Operations

```
fun listcontains :: 'a list => 'a => bool
where
    listcontains [] - = False
|   listcontains (a#A) b = ((a = b) | listcontains A b)

constdefs
    listintersect :: 'a list => 'a list => 'a list
    [simp]: listintersect A B == (filter (listcontains B) A)
```

B.8 Bound, Global and Free Names

Correctly determining the bound, free and global names of a function is essential to function definition and invocation as we have proposed them. The following is a list (adapted from [33]) of constructs that, if occurring in a function, bind the associated names locally:

- The **import** and **from** ... **import** ... statements.
- Function definitions, which bind the function name in the defining block.
- Assignment to names.
- The **del** statement.

Thus, searching for such constructs in a function body, collecting the names associated with them and removing explicitly global names gives all the names that should be considered locally bound. (Formal parameters are treated separately.) For the search and collection tasks, we define an Isabelle function, *boundrec*, with two helper functions, *bound-targets* and *bound-import*. The *bound-targets* function collects the names from a list of a targets, and *bound-import* collects the local names from a list of absolute-local name pairs. Since all of the binding constructs are statements, *boundrec* need not consider expressions; non-sequential definition is used to ensure that all types of statements are indeed accounted for. The definitions of the three functions follow:

```
fun bound-targets :: pythontarget list => string list
where
    bound-targets [] = []
|   bound-targets (t # tlist) = (case t of
        tAssName n => n # bound-targets tlist
        |tDelName n => n # bound-targets tlist
        |- => bound-targets tlist)
```

```
fun bound-import :: (string*string) list => string list
where
    bound-import [] = []
|   bound-import (first # rest) = (case first of
        (-, n) => n # bound-import rest)

function boundrec :: pythonstmt => string list
where
    boundrec (sIf t tlist els) = (case tlist of
        [] => (case t of (-, s) => boundrec s @ boundrec els)
        |t1#t2n => (case t1 of (-, s) =>
            boundrec s @ boundrec (sIf t t2n els)))
— recur on the tail of the test list by constructing a new sIf node
|   boundrec (sStmt s slist) = (case slist of
        [] => boundrec s
        |s1#s2n =>
            boundrec s1 @ boundrec (sStmt s s2n))
— recur on the tail of the statement list by constructing a new sStmt node
|   boundrec (sWhile - sb se) = boundrec sb @ boundrec se
|   boundrec (sAssign targs -) = bound-targets targs
|   boundrec (sFrom - pairlist) = bound-import pairlist
|   boundrec (sImport pairlist) = bound-import pairlist
|   boundrec (sFunction n - -) = [n]
|   boundrec sBreak = []
|   boundrec sContinue = []
|   boundrec (sDiscard -) = []
|   boundrec (sGlobal -) = []
|   boundrec sPass = []
|   boundrec (sPrint -) = []
|   boundrec (sPrintnl -) = []
|   boundrec (sReturn -) = []
by pat-completeness auto
termination by lexicographic-order
```

The explicitly global names of a function body are easier to determine, because it is simply a matter of recurring on statements and collecting all the names that appear in **global** statements. The following function performs this task:

```
function globalnames :: pythonstmt => string list
where
    globalnames (sGlobal nlist) = nlist
|   globalnames (sStmt s slist) = (case slist of
        [] => globalnames s
        |(s1#s2n) =>
            globalnames s1 @ globalnames (sStmt s s2n))
```

```
|   globalnames (sIf t tlist els) = (case tlist of
        [] => (case t of (-, s) => globalnames s @ globalnames els)
        |t1#t2n => (case t1 of (-, s) =>
            globalnames s @ globalnames (sIf t t2n els)))
|   globalnames (sWhile - sb se) =
        globalnames sb @ globalnames se
|   globalnames (sAssign - -) = []
|   globalnames sBreak = []
|   globalnames sContinue = []
|   globalnames (sDiscard -) = []
|   globalnames (sFunction - - -) = []
|   globalnames (sFrom - -) = []
|   globalnames (sImport -) = []
|   globalnames sPass = []
|   globalnames (sPrint -) = []
|   globalnames (sPrintnl -) = []
|   globalnames (sReturn -) = []
by pat-completeness auto
termination by lexicographic-order
```

Now, if a name occurs in a function but it is not bound, explicitly global or a formal parameter, it is considered free. In order to determine the free names of a function, we define a function, *allnamesrec*, to collect all the names that occur in a function body. For the most part, this is a straight-forward task, but lambda expressions and nested functions present a difficulty. Our operational semantics requires that the free names of nested functions and lambda expressions be listed as occurring in the defining function, in order to ensure that resolution information is propogated. As an approximation, it is safe to say that *all* the names of nested function and lambda expressions occur in the defining function, as they could be made to occur by adding $\langle \mathbf{Discard} : \langle \mathbf{Name} : n \rangle \rangle$ statements (as dead code) in the defining function body. However, for the sake of precision, we define a helper constant, *free-helper*, that takes a list of all the names occurring in a function body, a list of the bound variables, a list of the global names and a list of the formal parameters, and returns a list of the free names. It uses Isabelle's *filter* function, and a new helper function, *listcontains*, to reduce the list of all names to a list of free names. The definitions follow:

```
constdefs
    free-helper :: string list => string list => string list => string list => string list
    [simp]: free-helper A B G P == filter
        (λx. ~ listcontains (B @ G @ P) x)
        A
```

Next, we define the following helper function, which collects all the names occurring in an expression:

```
function allnames-exp :: pythonexp => string list
where
    allnames-exp (eBop - e1 e2) = allnames-exp e1 @ allnames-exp e2
|   allnames-exp (eCallFunc f elist) = (case elist of
```

```
            [] => allnames-exp f
            |e1#e2n => allnames-exp e1 @ allnames-exp (eCallFunc f e2n))
    |   allnames-exp (eCompare e c clist) = (case clist of
            [] => (case c of (-, e') => allnames-exp e @ allnames-exp e')
            |(-, e1)#c2n =>
                allnames-exp e1 @ allnames-exp (eCompare e c c2n))
    |   allnames-exp (eGetattr e -) = allnames-exp e
    |   allnames-exp (eIntLit -) = []
    |   allnames-exp (eLambda formals e) =
            free-helper (allnames-exp e) [] [] formals
    |   allnames-exp (eName n) = [n]
    |   allnames-exp (eAnd e elist) = (case elist of
            [] => allnames-exp e
            |e1#e2n => allnames-exp e1 @ allnames-exp (eAnd e e2n))
    |   allnames-exp (eOr e elist) = (case elist of
            [] => allnames-exp e
            |e1#e2n => allnames-exp e1 @ allnames-exp (eOr e e2n))
    |   allnames-exp (eUop - e) = allnames-exp e
    by pat-completeness auto
    termination by lexicographic-order
```

Next, *allnamesrec* and its remaining helper functions are defined as follows:

```
    fun allnames-targets :: pythontarget list => string list
    where
        allnames-targets [] = []
    |   allnames-targets (t # tlist) = (case t of
            tAssName n => n # allnames-targets tlist
            |tDelName n => n # allnames-targets tlist
            |tAssAttr e n => allnames-exp e @ allnames-targets tlist
            |tDelAttr e n => allnames-exp e @ allnames-targets tlist)

    fun allnames-exp-list :: pythonexp list => string list
    where
        allnames-exp-list [] = []
    |   allnames-exp-list (e1#e2n) = allnames-exp e1 @ allnames-exp-list e2n

    function allnamesrec :: pythonstmt => string list
    where
        allnamesrec (sAssign targs e) =
            (allnames-targets targs) @ (allnames-exp e)
    |   allnamesrec sBreak = []
    |   allnamesrec sContinue = []
    |   allnamesrec (sDiscard e) = allnames-exp e
    |   allnamesrec (sFrom - symlist) = bound-import symlist
    |   allnamesrec (sFunction n params body) = n #
            free-helper
```

```
                (allnamesrec body)
                (boundrec body)
                (globalnames body)
                params
|   allnamesrec (sGlobal -) = []
|   allnamesrec (sIf t tlist els) = (case tlist of
        [] => (case t of (e, s) =>
                allnames-exp e
                @ allnamesrec s
                @ allnamesrec els)
        |t1#t2n => (case t1 of (e, s) =>
                allnames-exp e
                @ allnamesrec s
                @ allnamesrec (sIf t t2n els)))
|   allnamesrec (sImport mlist) = bound-import mlist
|   allnamesrec sPass = []
|   allnamesrec (sPrint elist) = allnames-exp-list elist
|   allnamesrec (sPrintnl elist) = allnames-exp-list elist
|   allnamesrec (sReturn e) = allnames-exp e
|   allnamesrec (sStmt s slist) = (case slist of
        [] => allnamesrec s
        |s1#s2n => allnamesrec s1 @ allnamesrec (sStmt s s2n))
|   allnamesrec (sWhile e sb se) =
        allnames-exp e @ allnamesrec sb @ allnamesrec se
by pat-completeness auto
termination by lexicographic-order
```

Finally, we define some wrapper constants that present a more convenient interface to the above functions. For the sake of space efficiency, these wrappers use Isabelle's *remdups* function to remove duplicates from the lists. (If it were necessary to prove a theorem about these functions, it might be more convenient to leave out the *remdups* invocation, since it could complicate a proof.) Each wrapper is applied to a function's body and its list of formal parameters; the formal parameters are intentionally excluded from the lists returned by the wrappers, for the sake of space efficiency. The definitions of the wrappers follow:

```
constdefs
    boundnames :: pythonstmt => string list => string list
    boundnames body params == remdups (filter
        (λx. ~ listcontains (globalnames body @ params) x)
        (boundrec body))
    freenames :: pythonstmt => string list => string list
    freenames body params == remdups
        (free-helper
            (allnamesrec body)
            (boundrec body)
            (globalnames body)
```

params)

B.9 Transition Rules

constdefs — Comments for missing cases.
HOOK :: *string* => *'a*
[*simp*]: *HOOK s* == *undefined*

fun *TruthValue* :: *semvalue* => *bool*
where
TruthValue (*uInteger i*) = (*i* ~= *0*)
| *TruthValue* (*uBoolean b*) = *b*
| *TruthValue uNone* = *False*
| *TruthValue* - = *True*

constdefs — shorthand notation, page 28
StoreAndBind :: *store* ⇒
envhandle ⇒ *string* ⇒
semvalue ⇒ *store*
[*simp*]: *StoreAndBind S v n u* == (*case ApplyStoreEnv S v of*
Some E =>
(*ExtendStoreEnv*
(*ExtendStoreLoc S* (*Location v n*) *u*)
v
(*ExtendEnv E n* (*Location v n*))))

fun *CreateLocalsRec* ::
(*store*∗*environment*) => *envhandle* => *string list* => (*store*∗*environment*)
— helper for 𝕃, page 38
— equivalent formulation, not tail recursive as 𝕃 is
where
CreateLocalsRec (*S*, *E*) - [] = (*S*, *E*)
| *CreateLocalsRec* (*S*, *E*) *ε* (*first* # *rest*) =
(*case CreateLocalsRec* (*S*, *E*) *ε rest of* (*S′*, *E′*) =>
(*DelStore S′* (*Location ε first*),
(*ExtendEnv E′ first* (*Location ε first*))))

constdefs — 𝕃, page 38
CreateLocals :: *store* => *environment* => *string list* => *envhandle* => *store*
[*simp*]: *CreateLocals S E nlist v* == (*case CreateLocalsRec* (*S*, *E*) *v nlist of*
(*S′*, *E′*) => *ExtendStoreEnv S′ v E′*)

```
— All transition rules that trigger with a semantic value in C
function ExecRuleMTu ::
    (string ~=> pythonmodule) => envhandle
    => semvalue => envhandle => envhandle => continuation list => store
    => CEKS
where
    Cassattr2: ExecRuleMTu - - uL vL vG (AssAttrCont n uR # K) S =
    (case uL of
    uModule ε =>
        (case ApplyStoreEnv S ε of Some E =>
        (case ApplyEnv E n of
            Some l => uState uR vL vG K (ExtendStoreLoc S l uL)
            |None => uState uR vL vG K
                (StoreAndBind S ε n uR)))
    |- => HOOK "more general support for attributes")
|   Cassign: ExecRuleMTu - - u vL vG (AssignCont tlist # K) S =
    (case tlist of [] => uState uNone vL vG K S | (t1 # t2n) =>
    (case t1 of
    tAssName n => (if n = "None" then undefined else
        (case ApplyStoreEnv S vL of
        Some E => (case ApplyEnv E n of
            Some l =>
                uState u vL vG
                (AssignCont t2n # K)
                (ExtendStoreLoc S l u)
            |None => (case ApplyStoreEnv S vG of
                Some EG =>
                    uState u vL vG
                    (AssignCont t2n # K)
                    (case ApplyEnv EG n of
                    Some lG => (ExtendStoreLoc S lG u)
                    |None => StoreAndBind S vG n u)))))
    |tDelName n => (if n = "None" then undefined else
        (case ApplyStoreEnv S vL of
        Some E => (case ApplyEnv E n of
            Some l => (case ApplyStoreLoc S l of
                Some - => uState uNone vL vG K (DelStore S l))
            |None => (case ApplyStoreEnv S vG of
                Some EG => (case ApplyEnv EG n of
                    Some l => (case ApplyStoreLoc S l of
                        Some - => uState uNone vL vG K (DelStore S l)))))))
    |tAssAttr e n => (if n = "None" then undefined
        else eState e vL vG (AssAttrCont n u # K) S)
    |tDelAttr e n => (if n = "None" then undefined
        else eState e vL vG (DelAttrCont n u # K) S)))
```

```
|  Cbopleft: ExecRuleMTu - - u vL vG (BopLeftCont fbop eR # K) S =
      eState eR vL vG (BopRightCont fbop u # K) S
|  Cbopright: (ExecRuleMTu - - uR vL vG (BopRightCont bop uL # K) S) =
      (case BopEval bop uL uR of
          Some x => uState x vL vG K S
          |None => HOOK "inoperable data")
|  Ccallfunc: ExecRuleMTu - - u vL vG (CallFuncCont elist # K) S =
      (case u of
      (uClosure code parmnames localnames vR globalenv) =>
          (case ApplyStoreEnv S vR of Some ER =>
          (let vT=FreshEnvHandle S in
              uState uNone vL vG
                  (CallFuncTwoCont parmnames elist code vT globalenv # K)
                  (CreateLocals S ER localnames vT)))
      |- => HOOK "general callable objects")
|  Ccallfunc2: ExecRuleMTu - - - vL vG
                  (CallFuncTwoCont parmnames elist s vL' vG' # K) S =
      (case parmnames of
      [] => (case elist of
          [] => sState s vL' vG'
              (InvocationCont # RestoreEnvCont vL vG # K) S
          |- => HOOK "too many actual parameters")
      |n1 # n2n => (case elist of
          e1 # e2n => eState e1 vL vG
              (ParamBindCont n1 vL'
               # CallFuncTwoCont n2n e2n s vL' vG'
               # K) S
          |- => HOOK "too few actual parameters"))
|  Ccompare: ExecRuleMTu - - u vL vG (CompareCont c clist # K) S =
      eState (snd c) vL vG (CompareTwoCont u (fst c) clist # K) S
|  Ccompare2: ExecRuleMTu - - uR vL vG
              (CompareTwoCont uL cmpop1 clist # K) S =
      (case clist of
      [] => uState (uBoolean (CmpopEval cmpop1 uL uR)) vL vG K S
      |(cmpop2, e) # crest =>
          (if (CmpopEval cmpop1 uL uR)
          then eState e vL vG (CompareTwoCont uR cmpop2 crest # K) S
          else uState (uBoolean False) vL vG K S))
|  Cdelattr2: ExecRuleMTu - - u vL vG (DelAttrCont n u' # K) S =
      (case u of
      uModule ε =>
          (case ApplyStoreEnv S ε of Some E =>
          (case ApplyEnv E n of
          Some l => (case ApplyStoreLoc S l of
              Some - => uState u' vL vG K (DelStore S l)
```

```
                    |- => HOOK ''attribute not initialized'')
                |- => HOOK ''attribute does not exist''))
            |- => HOOK ''general attributes'')
|   Cdiscard: ExecRuleMTu - - - vL vG (DiscardCont # K) S =
        uState uNone vL vG K S
|   Cfrom: ExecRuleMTu - - - vL vG (FromCont vS npairs # K) S =
        (case npairs of
        [] => uState uNone vL vG K S
        |(nthere, nhere) # nrest =>
            (case ApplyStoreEnv S vS of Some ES =>
            (case ApplyEnv ES nthere of
            Some lthere =>
                (case ApplyStoreLoc S lthere of
                Some u =>
                    (case ApplyStoreEnv S vL of Some - =>
                        uState uNone vL vG (FromCont vS nrest # K)
                        (StoreAndBind S vL nhere u))
                |- => HOOK ''name not initialized'')
            |- => HOOK ''name not found'')))
|   Cgetattr: ExecRuleMTu - - u vL vG (GetAttrCont n # K) S =
        (case u of
        uModule ε =>
            (case ApplyStoreEnv S ε of Some E =>
            (case ApplyEnv E n of
            Some l =>
                (case ApplyStoreLoc S l of
                    Some u' => uState u' vL vG K S
                    |- => HOOK ''attribute not initialized'')
            |- => HOOK ''attribute not found''))
        |- => HOOK ''general attribute lookup'')
|   Cifcases: ExecRuleMTu - - u vL vG (IfCont body tests else-s # K) S =
        (if TruthValue u
        then (sState body vL vG K S)
        else (case tests of
            [] => sState else-s vL vG K S
            |test1#test2n =>
                eState (fst test1) vL vG
                    (IfCont (snd test1) test2n else-s # K) S))
|   Cimport: ExecRuleMTu MT vN - vL vG
            (ImportCont ((realname,localname)#modlist) # K) S =
(case ApplyStoreEnv S vN of Some EN =>
(case (ApplyEnv EN realname) of
    None => (let ε=FreshEnvHandle S ;
            S'= (StoreAndBind
                (ExtendStoreEnv S ε EmptyEnv)
```

```
                vN
                realname
                (uModule ε))
        in
        (case MT realname of
        Some m =>
            mState m ε ε
                (RestoreEnvCont vL vG
                 # ImportTwoCont localname ε
                 # ImportCont modlist # K)
                S'
        |- => HOOK "module not found"))
    |Some realnameInMod =>
        (case ApplyStoreLoc S realnameInMod of Some uM =>
        (case uM of uModule ε =>
            uState uNone vL vG (ImportTwoCont localname ε # ImportCont modlist # K) S))))
|   Cimportnil: ExecRuleMTu - - - vL vG (ImportCont [] # K) S =
        uState uNone vL vG K S
|   Cimport2: ExecRuleMTu - - u vL vG (ImportTwoCont n v # K) S =
        uState (uModule v) vL vG (AssignCont [tAssName n] # K) S
|   Cinvocation: ExecRuleMTu - - u vL vG (InvocationCont # K) S =
        uState uNone vL vG K S
|   Cparambind: ExecRuleMTu - - u vL vG (ParamBindCont n vT # K) S =
        (case ApplyStoreEnv S vT of Some - =>
            uState uNone vL vG K
                (StoreAndBind S vT n u))
|   Cprint: ExecRuleMTu - - - vL vG (PrintCont elist # K) S =
        (case elist of
        [] => uState uNone vL vG K S
        |e1#e2n => eState e1 vL vG (PrintCont e2n # K) S)
|   Cprintnl: ExecRuleMTu - - - vL vG (PrintnlCont elist # K) S =
        (case elist of
        [] => uState uNone vL vG K S
        |e1#e2n => eState e1 vL vG (PrintnlCont e2n # K) S)
|   Crestoreenv: ExecRuleMTu - - u vL vG (RestoreEnvCont vL' vG' # K) S =
        uState u vL' vG' K S
|   Cscand:
    ExecRuleMTu - - u vL vG (ScAndCont elist # K) S =
        (case elist of
        e1 # e2n =>
            (if (TruthValue u)
            then eState e1 vL vG (ScAndCont e2n # K) S
            else uState u vL vG K S)
        |[] => uState u vL vG K S)
|   Cscor:
```

```
    ExecRuleMTu - - u vL vG (ScOrCont elist # K) S =
        (case elist of
        e1 # e2n =>
            (if (TruthValue u)
            then uState u vL vG K S
            else eState e1 vL vG (ScOrCont e2n # K) S)
        |[] => uState u vL vG K S)
|   Cstmt: ExecRuleMTu - - - vL vG (StmtCont slist # K) S =
        (case slist of
        [] => uState uNone vL vG K S
        |s1#s2n => sState s1 vL vG (StmtCont s2n # K) S)
|   Cuop: ExecRuleMTu - - u vL vG (UopCont uop # K) S =
        (case UopEval uop u of
        Some res => uState res vL vG K S
        |- => HOOK "inoperable data")
|   Cwhiletest: ExecRuleMTu - - u vL vG (WhileTestCont test body else-s # K) S =
        (if (TruthValue u)
            then (sState body vL vG (WhileRunCont test body else-s # K) S)
            else (sState else-s vL vG K S))
|   Cwhilerun: ExecRuleMTu - - - vL vG (WhileRunCont test body else-s # K) S =
        eState test vL vG (WhileTestCont test body else-s # K) S
|   ExecRuleMTu - - u vL vG [] S = uState u vL vG [] S
by pat-completeness auto
termination by lexicographic-order


— All transition rules that trigger with a statement in C.
function ExecRuleMTs ::
    (string ~=> pythonmodule) => envhandle
    => pythonstmt => envhandle => envhandle => continuation list => store
    => CEKS
where
    Rassign: ExecRuleMTs - - (sAssign ts e) vL vG K S =
        eState e vL vG (AssignCont ts # K) S
|   Rbreak: ExecRuleMTs - - sBreak vL vG K S =
        (case K of K1 # K2n =>
        (case K1 of
            WhileRunCont - - - => uState uNone vL vG K2n S
            |InvocationCont => undefined
            |- => sState sBreak vL vG K2n S))
|   Rcontinue: ExecRuleMTs - - sContinue vL vG K S =
        (case K of K1 # K2n =>
        (case K1 of
            WhileRunCont - - - => uState uNone vL vG K S
            |InvocationCont => undefined
```

```
                |- => sState sContinue vL vG K2n S))
|   Rdiscard: (ExecRuleMTs - - (sDiscard e) vL vG K S) =
        eState e vL vG (DiscardCont # K) S
|   Rfrom: ExecRuleMTs MT vN (sFrom n npairlist) vL vG K S =
(case ApplyStoreEnv S vN of Some EN =>
(case ApplyEnv EN n of
    None => (let ε = FreshEnvHandle S in
        (case MT n of
        Some m =>
            mState m ε ε
                (RestoreEnvCont vL vG # FromCont ε npairlist # K)
                (ExtendStoreLoc
                    (ExtendStoreEnv
                        (ExtendStoreEnv S ε EmptyEnv)
                        vN
                        (ExtendEnv EN n (Location vN n)))
                    (Location vN n)
                    (uModule ε))
        |None => HOOK "module not found"))
    |Some lmod =>
        (case ApplyStoreLoc S lmod of Some (uModule ε) =>
            uState uNone vL vG (FromCont ε npairlist # K) S)))
|   Rfunction: ExecRuleMTs - - (sFunction funcname parmnames body) vL vG K S =
    (case ApplyStoreEnv S vL of Some EL =>
    (case listintersect (globalnames body) parmnames of
    [] => (let ε = (FreshEnvHandle S) in
        uState (uClosure body parmnames (boundnames body parmnames) ε vG)
        vL vG
        (AssignCont [tAssName funcname] # K)
        (ExtendStoreEnv S ε
            (ProjectEnv EL (freenames body parmnames))))
    |- => HOOK "syntax error: globals conflict with parameters"))
|   Rglobal: ExecRuleMTs - - (sGlobal glist) vL vG K S =
        uState uNone vL vG K S
|   Rif: ExecRuleMTs - - (sIf t tlist else-s) vL vG K S =
        eState (fst t) vL vG (IfCont (snd t) tlist else-s # K) S
|   Rimport: ExecRuleMTs - - (sImport modlist) vL vG K S =
        uState uNone vL vG (ImportCont modlist # K) S
|   Rpass: (ExecRuleMTs - - sPass vL vG K S) =
        uState uNone vL vG K S
|   Rprint: ExecRuleMTs - - (sPrint elist) vL vG K S =
        uState uNone vL vG (PrintCont elist # K) S
|   Rprintnl: ExecRuleMTs - - (sPrintnl elist) vL vG K S =
        uState uNone vL vG (PrintnlCont elist # K) S
|   Rreturn: ExecRuleMTs - - (sReturn e) vL vG K S =
```

```
        (case K of
        InvocationCont # K2n => eState e vL vG K2n S
        |- # K2n => sState (sReturn e) vL vG K2n S)
|   Rstmt: (ExecRuleMTs - - (sStmt s slist) vL vG K S) =
        sState s vL vG (StmtCont slist # K) S
|   Rwhile: ExecRuleMTs - - (sWhile test body else-s) vL vG K S =
        eState test vL vG (WhileTestCont test body else-s #K) S
by pat-completeness auto
termination by lexicographic-order
```

— All transition rules that trigger with an expression in C.

```
function ExecRuleMTe ::
    pythonexp => envhandle => envhandle => continuation list => store
    => CEKS
where
    Rbop: (ExecRuleMTe (eBop bop eL eR) vL vG K S) =
        eState eL vL vG (BopLeftCont bop eR # K) S
|   Rcallfunc: ExecRuleMTe (eCallFunc ef elist) vL vG K S =
        eState ef vL vG (CallFuncCont elist # K) S
|   Rcompare: ExecRuleMTe (eCompare e cmp cmplist) vL vG K S =
        eState e vL vG (CompareCont cmp cmplist # K) S
|   Rgetattr: ExecRuleMTe (eGetattr e n) vL vG K S =
        eState e vL vG (GetAttrCont n # K) S
|   Rintlit: (ExecRuleMTe (eIntLit i) vL vG K S) =
        uState (uInteger i) vL vG K S
|   Rlambda: ExecRuleMTe (eLambda nlist e') vL vG K S =
    (case ApplyStoreEnv S vL of Some EL =>
    (let ε = (FreshEnvHandle S) in
        uState (uClosure (sReturn e') nlist [] ε vG)
            vL vG K
            (ExtendStoreEnv S ε
                (ProjectEnv EL (freenames (sReturn e') nlist)))))
|   Rname: ExecRuleMTe (eName n) vL vG K S =
    (if n = ''None''
    then (uState uNone vL vG K S)
    else
        (case ApplyStoreEnv S vL of Some EL =>
        (case ApplyEnv EL n of
            None =>
                (case ApplyStoreEnv S vG of Some EG =>
                (case ApplyEnv EG n of
                Some nloc =>
                    (case ApplyStoreLoc S nloc of
                    Some u => uState u vL vG K S
```

```
                    |None => HOOK ''uninitialized name'')
                |None => HOOK ''name not found''))
            |Some nloc => (case (ApplyStoreLoc S nloc) of
                Some u => uState u vL vG K S))))
|   Rscand: (ExecRuleMTe (eAnd e elist) vL vG K S) =
        eState e vL vG (ScAndCont elist # K) S
|   Rscor: (ExecRuleMTe (eOr e elist) vL vG K S) =
        eState e vL vG (ScOrCont elist # K) S
|   Ruop: ExecRuleMTe (eUop uop expr) vL vG K S =
        eState expr vL vG (UopCont uop # K) S
by pat-completeness auto
termination by lexicographic-order
```

— Speed up and parallelize the admission of the transition rules by splitting them into seperate functions.

```
function ExecRuleMT :: (string ~=> pythonmodule) => envhandle => CEKS => CEKS
where
    ExecRuleMT MT vN (uState u vL vG K S) = ExecRuleMTu MT vN u vL vG K S
|   ExecRuleMT MT vN (sState s vL vG K S) = ExecRuleMTs MT vN s vL vG K S
|   ExecRuleMT - - (eState e vL vG K S) = ExecRuleMTe e vL vG K S
|   Rmodule: ExecRuleMT - - (mState m vL vG K S) =
        (case m of mModule s => sState s vL vG K S)
by pat-completeness auto
termination by lexicographic-order

constdefs
    ExecRule :: CEKS => CEKS
    [simp]: ExecRule == ExecRuleMT (%x. None) arbitrary
```

Appendix C

An Invariant of IntegerPython

The following is the proof of the invariant described in Section 4.3, that for a state, *M*, satisfying *SNInvariant M*, the next state, *ExecRule M*, is either *undefined* (indicating a stuck machine) or it satisfies *SNInvariant* (*ExecRule M*):

declare *Let-def* [*simp*]

theorem *inv-SameName*:
assumes *SNInvariant M*
shows *ExecRule M* = *undefined* | *SNInvariant* (*ExecRule M*)
proof (*cases M*)
    case (*mState m vL vG K S*)
    show *?thesis* **using** *prems*
        by (*auto split*: *pythonmodule.split*)
next
    case (*eState e vL vG K S*)
    from *prems* **have** *preinvL*: *SameName* (*ApplyStoreEnv S vL*) **by** *simp*
    show *?thesis* **using** *prems*
    proof (*cases e*)
        case (*eLambda nlist e'*)
        show *?thesis* **using** *prems samename-project preinvL*
            by (*cases ApplyStoreEnv S vL*) (*simp-all split*: *list.split*)
    qed (*auto split*: *list.split option.split*)
next
    case (*sState s vL vG K S*)
    from *prems* **have** *preinvL*: *SameName* (*ApplyStoreEnv S vL*) **by** *simp*
    show *?thesis* **using** *prems*
    proof (*cases s*)
        case (*sFunction f P b*)
        show *?thesis* **using** *prems samename-project preinvL*
            by (*cases ApplyStoreEnv S vL*) (*simp-all split*: *list.split*)
    qed (*auto split*: *list.split option.split semvalue.split continuation.split*)
next

```
case (uState u vL vG K S)
from prems have preinvL: SameName (ApplyStoreEnv S vL) by simp
from prems have preinvG: SameName (ApplyStoreEnv S vG) by simp
show ?thesis using prems
proof (cases K)
    case (Cons K1 K2n) show ?thesis using prems
    proof (cases K1)
        case (ParamBindCont n v)
        from prems have preinv: SameName (ApplyStoreEnv S v)
            by simp
        with prems show ?thesis
            by (cases ApplyStoreEnv S v) simp-all
    next
        case (GetAttrCont n) show ?thesis using prems
            by (auto split: semvalue.split option.split)
    next
        case (DelAttrCont n) show ?thesis using prems
        proof (cases u)
            case (uModule v)
            from prems have preinv:
                SameName (ApplyStoreEnv S v) by simp
            with prems show ?thesis
                by (auto split: option.split)
        qed simp-all
    next
        case CallFuncCont show ?thesis using prems
        proof (cases u)
            case (uClosure - - - vL' -)
            have preinv: SameName (ApplyStoreEnv S vL')
                using prems by simp
            with prems preinvL preinvG samename-cl show ?thesis
                by (cases ApplyStoreEnv S vL') simp-all
        qed simp-all
    next
        case (ImportCont list) show ?thesis using prems
        proof (cases list)
            case (Cons M1 -) show ?thesis using prems
            proof (cases M1)
                case (Pair realname localname)
                show ?thesis using prems preinvL preinvG
                    by (auto split: semvalue.split option.split)
            qed
        qed simp
    next
        case (AssignCont targets) show ?thesis using prems
```

```
    proof (cases targets)
        case (Cons t1 -) show ?thesis using prems
        proof (cases t1)
            case (tAssName n)
            show ?thesis using prems
            proof (cases ApplyStoreEnv S vL)
                case (Some E)
                thus ?thesis using prems
                proof (cases ApplyEnv E n)
                    case None
                    show ?thesis using prems preinvL preinvG
                    proof (cases ApplyStoreEnv S vG)
                        case (Some EG)
                        from prems preinvL preinvG show ?thesis
                            by (auto split: option.split)
                    qed simp
                qed simp
            qed simp
        next
            case (tDelName n)
            show ?thesis using prems
            proof (cases ApplyStoreEnv S vL)
                case (Some EL)
                show ?thesis using prems
                proof (cases ApplyEnv EL n)
                    case (Some l) show ?thesis using prems
                    by (cases ApplyStoreLoc S l) simp-all
                next
                    case None show ?thesis using prems
                    proof (cases ApplyStoreEnv S vG)
                        case (Some EG)
                        show ?thesis using prems
                        proof (cases ApplyEnv EG n)
                            case (Some l)
                            show ?thesis using prems preinvG
                            by (simp split: option.split)
                        qed simp
                    qed simp
                qed
            qed simp
        qed simp-all
    qed simp
next
    case (FromCont v npairs) show ?thesis using prems
    proof (cases npairs)
```

```
                case (Cons first rest)
                show ?thesis using prems
                proof (cases first)
                    case (Pair realname localname)
                    show ?thesis using prems
                    proof (cases ApplyStoreEnv S v)
                        case (Some ES) show ?thesis using prems
                        proof (cases ApplyEnv ES realname)
                            case (Some lS) show ?thesis using prems
                            proof (cases ApplyStoreLoc S lS)
                                case (Some umod)
                                show ?thesis using prems preinvL
                                    by (cases ApplyStoreEnv S vL)
                                        simp-all
                            qed simp
                        qed simp
                    qed simp
                qed
            next
                case (AssAttrCont n uR) show ?thesis using prems
                proof (cases u)
                    case (uModule ε)
                    from prems have preinv:
                        SameName (ApplyStoreEnv S ε) by simp
                    show ?thesis using prems
                    proof (cases ApplyStoreEnv S ε)
                        case (Some E)
                        show ?thesis using prems preinv
                        by (cases ApplyEnv E n) auto
                    qed simp
                qed simp-all
            qed (auto split: option.split list.split prod.split)
        qed simp
qed

declare Let-def [simp del]
```