ADVICE ON STRUCTURING COMPILER BACK ENDS AND PROVING THEM CORRECT

A Thesis

Presented to the Faculty of the Graduate School

of the College of Computer Science

of Northeastern University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Dino P. Oliva

January 1994

Abstract

In this dissertation we present a methodology for the verification of language implementations based on formal semantics. While this is not a new concept, the methodology presented is interesting for several reasons:

- It is easy to use
- It can be applied to languages specified by traditional denotational techniques
- It mirrors traditional language implementation schemes
- It is powerful enough to generate code for contemporary machine architectures

Our approach advocates the use of denotational semantics to verify compiler front ends (machine independent transformations) and operational semantics to verify compiler back ends (machine dependent transformations).

To illustrate the methodology, we present and verify an implementation of the language ν-LISP, a dialect of the Scheme programming language [13]. ν-LISP is restricted to obey stack discipline which introduces a number of issues we must deal with to ensure that the language is implemented properly (i.e. using only a stack). We begin by presenting the formal definition of ν-LISP using denotational semantics. We then derive and verify a compiler front end for the language using denotational reasoning. At this point, we switch to operational reasoning to characterize a class of back ends that will correctly generate code for our target machine, a machine model based upon contemporary RISC architectures. Finally, we present a compiler back end and verify its correctness by demonstrating that it falls within the previously characterized class of correct back ends.

Acknowledgements

The best things in life are not free but, rather, are beyond money; their price is agony and sweat and devotion ... and the price demanded for the most precious of all things in life is life itself — ultimate cost for perfect value.

— Robert A. Heinlein

I would like to thank my advisor, Mitchell Wand, for his years of guidance, patience, and good humor. This dissertation is dedicated to him, and all others who have taken on the arduous task of advising.

I would also like to thank my committee, Richard Kelsey, Karl Lieberherr, and John Ramsdell, each of whom had a great impact upon my views of programming language design and implementation. Their insights into my own research added immeasurably to its quality.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this dissertation, we present a methodology for verified language implementation based upon denotational and operational semantics. Our ultimate goal was to derive techniques simple to apply yet powerful enough to verify the correctness of efficient language implementations. To this end, we have drawn from formal language semantics and practical insight gained from existing semantics-based language implementations.

1.1 Formal semantics

Formal semantics are used to specify the definition of programming languages. The intention is to give implementors the insight they need to implement a language properly and the formalisms to verify the implementation's correctness. In practice, verification has turned out to be quite difficult. The reasons for this depends upon the formalism used to define the language. In what follows, we will discuss the two major formalisms and their respective difficulties with verification. We will then discuss our approach, which combines the two and capitalizes on the strength of each.

1.1.1 Operational semantics

In operational semantics, programming languages are characterized by the execution of an abstract machine. Correctness is verified by proving that the implementation preserves the execution sequence of the abstract machine. The problem is that different implementations of programming languages may follow radically different approaches. If the approach used by a particular implementation is different from the language semantics, there may well be no way of proving the correctness of the implementation.

1.1.2 Denotational semantics

In the early seventies, Scott developed the first mathematically-natural models of the λ-calculus. This gave rise to denotational semantics where each phrase in the language being defined has a meaning function associated with it. This function is defined in terms of the meaning functions associated with the phrase's sub-phrases. To verify a language implementation, it is necessary to show that, for each phrase, the implementation produces a function equal to its defined meaning[1]. Since the meanings of phrases in the source language are defined compositionally, correctness proofs can make use of structural induction on the source language. This approach overcomes the problems inherent with operational semantics because it doesn't matter how a function is defined, just that it is equal to the one defined by the semantics.

The problem with the denotational approach is that language implementations generate code for a target machine. This means that we must prove the functions defined by the language semantics are equal (or congruent) to functions defined by the target machine. However, before a congruence proof can be constructed, the domains of the source language and target machine have to be related. This poses a serious problem, one that was initially overcome by the use of *inclusive predicates*. These inclusive predicates worked as required for small languages but gave rise to very long and complex proofs. It was clear that better techniques were required to handle real programming language definitions.

[1]Here, two functions are considered equal if they return the same value on all inputs.

Wand [71] addressed this problem when he showed how to factor language definitions into combinators (abstractions with no free variables). Combinators have a well defined functionality and research has shown that they can be implemented efficiently in hardware [67, 11, 25, 64, 53]. Factoring language definitions into combinators gave a natural correspondence between source language semantics and the machine defined by the combinators. Since the combinators were derived directly from the source language semantics, the target machine was defined in the same semantics domains. Proving the equivalence of the original language definition and the derived one was quite simple because no inclusive predicates were necessary. Clinger [12] refined Wand's approach by showing a simple method for proving the congruence between a language definition and an alternative language definition specified using the combinator approach.

The methods developed by Wand and Clinger showed how to transform source language phrases into code for an abstract machine. The approach was very similar to the approach taken by traditional compiler front ends. To prove the correctness of a complete language implementation, the methods had to be extended to handle more concrete machines.

Wand and Montenyohl [75] and Wand and Wang [77] attempted to factor language definitions so that derived target machines looked like more traditional architectures. This approach led to a considerable increase in the complexity of the verification proofs. The reason for this was that they had to track and prove properties about steps in the machine execution. This suggested that operational semantics might be a better approach at this level.

1.1.3 Hybrid semantics

Researchers in operational semantics have put much work into defining languages and their appropriate abstract machines. They have also faced the problem of implementing abstract machines on traditional, concrete architectures. Their standard approach has been to write an interpreter for the abstract machine on the target machine.

Recently, an alternative approach has been explored by Hannan [22] whereby the code for the abstract machine was transformed into equivalent code for the underlying concrete machine. His work showed how to automatically transform a restricted class of abstract machines into concrete machines where the representations of data and code were explicit. Unfortunately, the class of machines he could handle was too restrictive to be of general use.

Based on this work, Wand and Oliva developed *storage layout relations* to formally specify the relationship between machines defined using operational semantics [79]. Using this technology, the abstract machines produced by the Wand-Clinger approach could be mapped to more concrete machines. The formalisms could then be employed to prove that as long as a concrete machine was in correspondence with an abstract machine, the concrete machine would simulate the abstract machine throughout execution. This work, however, only considered the representations of data. The abstract and concrete machines had to have the same instruction set.

In this dissertation, we address the problem of code representation by extending the use of storage layout relations. Once the correspondence between an abstract and concrete machine has been specified via these relations, the verification of concrete machine code can be performed in either of two ways:

1. A back end can be shown to always generate code that preserves the mappings specified by the storage layout relations. We will illustrate this approach in chapters 7 and 8.

2. Generated concrete machine code can be checked directly with the original abstract machine code to verify that it satisfies the storage layout relations. This approach allows the verification of code regardless of algorithm used to produce it (pre-existing back ends, for example).

1.2 Semantics-based language implementations

In this section, we discuss actual language implementations based upon semantic definitions. In each case, we discuss the merits and weaknesses of the approach used by the

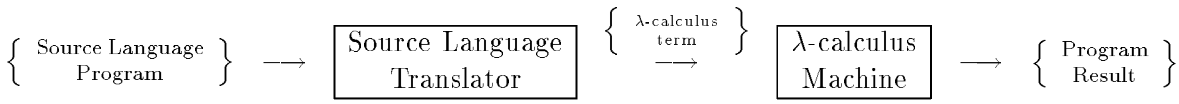

Figure 1: Verified Language Implementation

implementation.

1.2.1 Direct approaches

The straightforward approach to a verified language implementation is to implement a denotational semantics definition directly (see Figure 1). The traditional approach to a denotational definition of a programming language is to use the λ-calculus to define source language meanings. To implement such a definition directly, one must first implement a translator that transforms phrases from the source language into the associated λ-terms specified in the semantic definition. To verify the implementation, it must then be proved that the translator does, in fact, transform all phrases into the specified λ-terms.

The next step is the evaluation of the λ-terms. Since λ-calculus machines are generally not available, the usual course of action is to implement a λ-calculus interpreter. For a completely verified language implementation, this interpreter must also be verified.

Given the success of translating machine-readable, formal grammar specifications into parsers [26], researchers began to wonder if a similar approach could be taken with denotational specifications. Instead of producing parsers from grammar specifications, these programs would produce source language to λ-term translators from machine readable specifications. The generated translator could then be used in conjunction with the λ-calculus machine for a semantics-based language implementation (see Figure 2).

The direct approach to automated language implementation was explored by many researchers [41, 49, 56, 60, 74, 3]. Although the implementations they produced were not competitive with traditional techniques, they proved the viability of incorporating formal language definition into implementations. The insights these systems provided lead to the

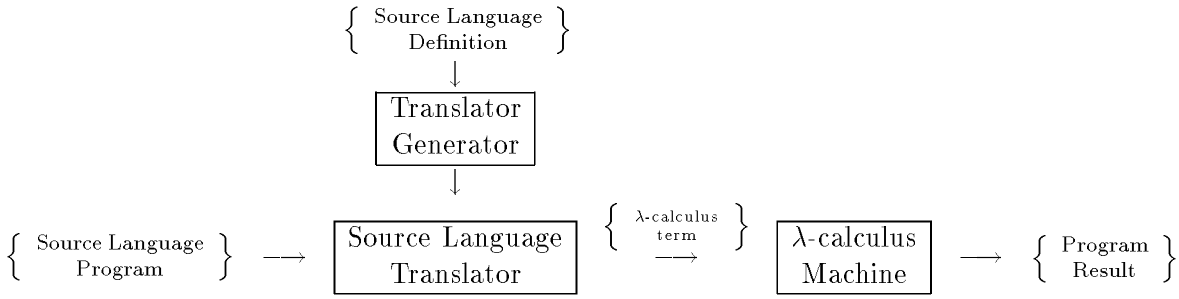

Figure 2: Automated Translator Generator

more advanced systems discussed in the next section. In the remainder of this section, we will summarize a cross-section of the early systems.

SIS

Mosses' Semantics Implementation System [41, 8] was the earliest well known system that used the direct approach. Language semantics were specified in DSL, a machine readable version of the λ-calculus. The system produced language implementations that would translate phrases from the defined (source) language into λ-calculus terms. At runtime, the λ-terms were executed by an interpreter performing β-reductions using a call-by-need strategy.

SIS was not meant to generate production language implementations and it had a number of drawbacks:

- It could only handle SLR grammars.
- It could only process small language definitions.
- It produced very inefficient language implementation.

Despite these drawbacks, SIS accomplished its goal by demonstrating that language specifications could be used to automatically generate language implementations.

PSP

Paulson's PSP system [49] was a much less restrictive system than SIS. It also specified language semantics using a machine readable form of the λ-calculus but the language implementations produced code for Landin's SECD machine [33]. Again, the generated compilers were not competitive with production compilers but his system proved that implementations could be generated for non-trivial language definitions[2].

SPS

Wand's Semantic Prototyping System [74] was notable because of its use of Scheme as a λ-calculus engine. Language definitions were specified using Scheme code as the machine readable λ-calculus. The generated language implementations translated source language phrases into Scheme code. Due to efficient Scheme implementations, the language implementations were useful for prototyping new language definitions but SPS was still not competitive with traditional implementations approaches.

1.2.2 Two-level approaches

It was not surprising that the direct approach to language implementation lead to inefficient implementations. With the insights gained from the direct approach, researchers began to focus on more sophisticated systems. The key insight gained from the early systems was that the language implementations had to include some differentiation between static (compile time) and dynamic (runtime) information.

The later approaches are characterized by their use of the two-level approach shown in Figure 3. The static component of the language definition transforms programs into an intermediate representation (or intermediate code). The dynamic component acts as a virtual machine that executes this intermediate code running on top of the λ-calculus.

[2]In [49], Paulson reports having generated implementations for Pascal and Fortran.

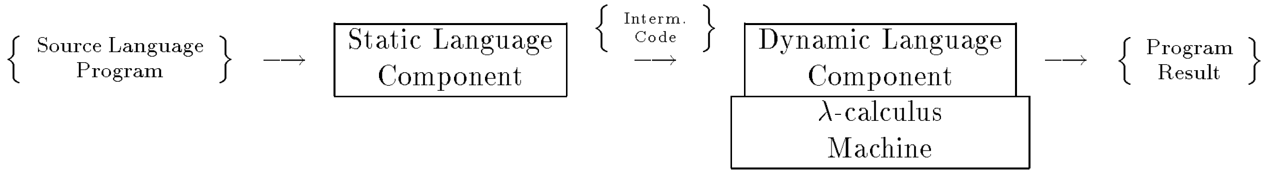

Figure 3: Two-Level Approach

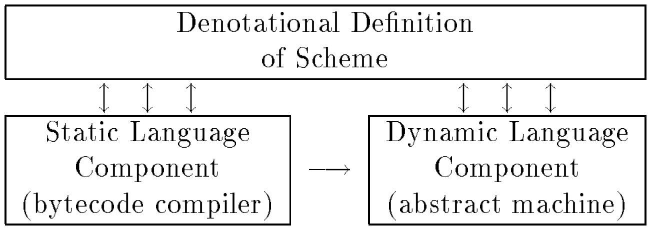

Figure 4: Scheme 311 Compiler Semantics

Scheme 311

Clinger [12] used the two-level approach in the implementation and verification of the Scheme 311 compiler. He did this by first giving a traditional semantic definition for Scheme. He then, by hand, extracted the static and dynamic properties of the language and gave an alternative, two-level semantics for Scheme using Wand's combinator approach. In this approach the static portion of the new Scheme definition translated source programs into combinator trees. These combinator trees formed an intermediate code which was then evaluated by the dynamic portion of the semantics. This dynamic portion formed an abstract machine for the intermediate code.

Research has shown that combinators can be implemented directly in hardware but the Scheme 311 compiler was meant to run on a more conventional architecture. Hence the combinator code for the abstract machine had to be executed by an interpreter for the abstract machine or translated into "native" code for the target architecture. Clinger's work does not address this issue.

TML

Rather than prove congruence between a traditional language definition and a two-level one, Nielson and Nielson [43, 44] advocated the use of two-level semantics for canonical language definitions. They developed a formal system for specifying two-level definitions and built a compiler generator system from these formalisms. While the system could produce compilers, it remains unclear how the system performs for non-trivial language definitions.

Mess

Lee and Pleban [34, 35, 36, 37] also advocated two-level language definitions over traditional language definitions. In their approach, language specifications were broken up into macro and micro semantics. The intention was that the macro semantics would encapsulate the compile time computations while the micro semantics would encapsulate the runtime computations. With respect to the two-level approach shown in Figure 3, the macro semantics corresponded to the static language component while the micro semantics corresponded to the dynamic language component.

Macro semantics were defined using action semantics [42] in which language semantics are defined using abstract operators (actions). Micro semantics gave the actual definition of the actions. There were several options available for specifying micro semantics. The λ-calculus could be used, so that the composition of the macro and micro semantics formed a denotational language definition. Alternatively, an abstract machine could be used, giving operational specifications.

Lee and Pleban developed the MESS system for generating compilers from macro and micro semantics. It worked much in the way that the earlier automated system did. However, for traditional languages, the two-level semantics approach gave much more encouraging results. They were able to generate a compiler for SOL/C, a subset of the C programming language. The generated compiler produced code that compared favorably to production compilers in both size and efficiency [37].

While the MESS system performed favorably for SOL/C, it remains unclear how it would perform for more complex languages. Furthermore, the work does not formally consider the relationship between the denotational micro semantics, the operational micro semantics, and the code generator.

Chapter 2

Overview

In the rest of the dissertation, we use the approach described in chapter 1 to prove the correctness of an implementation of the language ν-LISP, a dialect of Scheme inspired by Pure PreScheme [45]. The major contribution of this work is the verification of the compiler back end and its use of the underlying target machine. While the issue of code optimization is outside of the focus of this work, it could be integrated using pre-existing techniques [15, 80].

We begin by introducing some notational conventions in chapter 3. In chapter 4 we give a formal definition of ν-LISP using denotational semantics. This definition contains the usual continuation component to express flow of control and store component to express the imperative features of the language.

Using the Wand-Clinger approach, we then derive and verify a compiler front end for the language in chapter 5. The front end produces code for an abstract machine manipulating tree structures. The meaning of the abstract machine code is defined denotationally. This allows us to easily prove the correctness of the compiler front end with respect to the original language definition.

Our efforts now focus on the compiler back end. It is the job of the compiler back end to transform code for the abstract machine into equivalent code for the target machine. To prove the correctness of the compiler back end, we must verify that the code it produces causes the register machine to simulate the actions of the abstract machine on the original code. As this kind of reasoning is best done operationally, we change from denotational to operational reasoning by giving an equivalent, operational definition of the abstract machine. We specify our target machine, the register machine, operationally as well. The register machine is based upon contemporary RISC architectures. It contains a large set of general purpose registers, linear memory, and a linear instruction store referenced through an instruction pointer.

As a first step, in chapter 6 we consider the representation of the abstract machine's data in the register machine. We do this by presenting the stack machine, a hybrid machine that uses the same instruction set as the abstract machine but uses the data representations of the register machine (linear memory and pointers). Each instruction is defined to manipulate data representations in the stack machine analogous to the way it manipulates the data in the abstract machine. The representation strategies employed represent mutable variables directly on the stack rather than having an extra level of indirection through the store. We formalize the data representations using storage layout relations which allow us to verify their correctness.

In chapter 7, we consider representing the instructions of the abstract machine with the instructions of the register machine. This representation is complicated by several considerations:

1. The code in the register machine is a linear sequence of instructions whereas the code for the stack machine is tree structured. In transforming stack machine code into register machine code, this tree structure must be linearized properly.

2. Each instruction of the stack machine may need several instructions of the target machine to perform the same action.

3. Stack machine instructions implicitly manipulate data on the stack while the opcodes

explicitly manipulate data in registers. Hence the relationship between stack machine code and register machine code is dependent upon which parts of the stack are currently in registers.

Rather than write a compiler back end and verify its correctness, we instead provide a general set of criteria relating stack and register machine code. The relationship is parameterized by register assumptions which formalize how the stack is cached in registers. We then prove that any back end satisfying these criteria will transform code from the abstract machine into code for register machine while preserving the semantics.

We use these criteria for back ends in chapter 8 when we write a back end using a typical register protocol for passing parameters and a simple register allocator. We prove its correctness simply by demonstrating that it meets the criteria. While the register allocation scheme is simple, due to the generality of the correctness criteria, more sophisticated allocation schemes could be incorporated and verified using the same approach.

Chapter 3

Notation

In this dissertation, we assume a working knowledge of denotational semantics [58, 63] and operational semantics [21]. We rely heavily on the use of sequences and contours and will discuss them here.

3.1 Sequences

Similar to [13, 28], we use the following notational conventions for finite sequences:

$\langle\,\rangle$	empty sequence
t^*	a sequence of zero or more t's
t^+	a sequence of one or more t's
$s\,\S\,t$	sequence concatenation
$t :: t^*$	concatenation of a single element to a sequence
$t^* \downarrow n$	the nth element of sequence t^* (0-based)
$t^* \dagger n$	t^* with the first n elements removed
$\langle \ldots \rangle$	sequence formation (i.e. $t_1 ::\ \ldots\ :: t_n :: \langle\,\rangle = \langle t_1, \ldots, t_n \rangle$)
$\#t^*$	the length of sequence t^*
$\rightleftharpoons t^*$	the reverse of sequence t^*
$t^* \uparrow n$	the nth element from the end of sequence t^* (0-based)

In addition for sequences whose elements have an equality operation, we define the following operations:

- Equality[1]

$$(s^* =_S t^*) = \begin{cases} \text{if } s^* = \langle\,\rangle \text{ and } t^* = \langle\,\rangle \text{ then } \mathit{true} \\ \text{if } s^* = s :: s^{*\prime} \text{ and } t^* = t :: t^{*\prime} \text{ and } s = t \text{ then } s^{*\prime} =_S t^{*\prime} \\ \text{otherwise } \mathit{false} \end{cases}$$

- Membership

$$(t \in t^*) = \begin{cases} \text{if } t^* = t :: t^{*\prime} \text{ then } \mathit{true} \\ \text{if } t^* = t' :: t^{*\prime} \text{ then } t \in t^{*\prime} \\ \text{otherwise } \mathit{false} \end{cases}$$

- Index

$$t^*(t) = \begin{cases} \text{if } t^* = t :: t^{*\prime} \text{ then } 0 \\ \text{if } t^* = t' :: t^{*\prime} \text{ then } t^{*\prime}(t) + 1 \\ \text{otherwise } \mathit{error} \end{cases}$$

- 0 based substitution

$$t^*[t/n] = \begin{cases} \text{if } t^* = t' :: t^{*\prime} \text{ and } n = n' + 1 \text{ then } t' :: t^{*\prime}[t/n'] \\ \text{if } t^* = t' :: t^{*\prime} \text{ and } n = 0 \text{ then } t :: t^{*\prime} \\ \text{otherwise } \mathit{error} \end{cases}$$

3.2 Contours

A contours is a sequence whose elements are themselves sequences[2]. For readability, we introduce an alternative notation for sequence concatenation when the sequences are contours. Additionally, we overload those sequence operators that use an index to work with indices specified as (depth,offset) pairs.

- Concatenation

[1] We drop the subscript from $=_S$ when its usage is unambiguous.

[2] We call these contours because they are used to model lexical contours in environments, symbol tables, displays, etc.

$$t \cdot c = t :: c$$

- Access

$$c \downarrow (d,o) = \begin{cases} \text{if } c = (t^* \cdot c') \text{ and } d = 0 \text{ then } t^* \downarrow o \\ \text{if } c = (t^* \cdot c') \text{ and } d = d' + 1 \text{ then } c' \downarrow (d', o)) \\ \text{otherwise } \mathit{error} \end{cases}$$

- Substitution

$$c[a/(d,o)] = \begin{cases} \text{if } c = (t^* \cdot c') \text{ and } d = 0 \text{ then } (t^*[t/o] \cdot c') \\ \text{if } c = (t^* \cdot c') \text{ and } d = d' + 1 \text{ then } (t^* \cdot c'[a/(d', o)]) \\ \text{otherwise } \mathit{error} \end{cases}$$

- Similarity

$$(c \approx c') = \begin{cases} \text{if } c = \langle\,\rangle \text{ and } c' = \langle\,\rangle \text{ then } \mathit{true} \\ \text{if } c = (t^* \cdot c'') \text{ and } c' = (t^{*'} \cdot c''') \text{ and } \#t^* = \#t^{*'} \text{ then } c'' \approx c''' \\ \text{otherwise } \mathit{false} \end{cases}$$

For contours whose underlying sequence elements have an equality operation, we generalize the following operations:

- Membership

$$(t \in c) = \begin{cases} \text{if } c = (t^* \cdot c') \text{ and } t \in t^* \text{ then } \mathit{true} \\ \text{if } c = (t^* \cdot c') \text{ and } t \notin t^* \text{ then } t \in c' \\ \text{otherwise } \mathit{false} \end{cases}$$

- Index

$$c(t) = \begin{cases} \text{if } c = (t^* \cdot c') \text{ and } t \in t^* \text{ then } (0, t^*(t)) \\ \text{if } c = (t^* \cdot c') \text{ and } t \notin t^* \text{ and } c'(t) = (d,o) \text{ then } (d+1, o) \\ \text{otherwise } c = \langle\,\rangle \text{ then } \mathit{error} \end{cases}$$

- Equality[3]

[3]We drop the subscript from $=_C$ when its usage is unambiguous.

$$(c =_C c') = \begin{cases} \text{if } c = \langle\,\rangle \text{ and } c' = \mathit{nil} \text{ then } \mathit{true} \\ \text{if } c = (t^* \cdot c'') \text{ and } c' = (t^* \cdot c''') \text{ then } c'' =_C c''' \\ \text{otherwise } \mathit{false} \end{cases}$$

- Containment

$$(c \subseteq_C c') = \begin{cases} \text{if } c =_C c' \text{ then } \mathit{true} \\ \text{if } c = (t^* \cdot c'') \text{ then } c'' \subseteq_C c' \\ \text{otherwise } \mathit{false} \end{cases}$$

- Proper containment

$$(c \subset_C c') = \begin{cases} \text{if } c' = (t^* \cdot c'') \text{ and } c \subseteq_C c'' \text{ then } \mathit{true} \\ \text{otherwise } \mathit{false} \end{cases}$$

3.3 Equality as a binding form

For notational convenience, we use equality tests to perform bindings in the true branches of conditionals. In the following example the test for equality binds t and t^*:

$$(t^* = t :: t^{*'}) \to t^{*'}, \mathit{error}$$

Similarly, we use non-equality to bind values in the false branch of a conditional. Hence the previous example could also be expressed as:

$$(t^* \neq t :: t^{*'}) \to \mathit{error}, t^{*'}$$

Chapter 4

The Algorithmic Language ν-LISP

In this chapter we describe the programming language ν-LISP, a dialect of the Scheme programming language [13] restricted to obey stack discipline. Its design was greatly influenced by the language Pure PreScheme [45, 46] which was, in turn, influenced by the language PreScheme [29]. It consists of a small, extensible core with sequencing, conditional branches, assignment, procedure abstraction, and procedure application built in. Although recursion is not explicitly available, recursive procedure definitions can be added through a syntactic extension in the same way Scheme's `letrec` is (and similarly for `let`, `let*`, and so forth).

The enforcement of stack discipline requires a number of restrictions to Scheme. Most of these restrictions are imposed by the enforcement of the "no upward funargs" rule, which states that closures[1] may not contain environments not contained in the environment they are being returned to. We summarize these restrictions by describing how ν-LISP differs from Scheme:

- The values returned by procedures are checked. If the return value is a closure, its environment must be contained in the environment it is being returned to[2].

[1]Procedure representations containing the environment in which the procedure was created.

[2]This restriction prohibits the implementation of Scheme's `call/cc`. Alternatively, a restricted form of `call/cc`, similar to Common LISP's `catch` and `throw` mechanisms, could be added to the language.

- In order to ensure that no upward funargs are implicitly returned through an assignment, all assignments are checked. A closure may only be assigned to a variable if the closure's environment is contained within the environment that the variable is defined in.

- Primitives must also ensure that they never return upward funargs (as described section 4.6).

- Finally, arguments to procedures are restricted so that tail recursive calls can be implemented efficiently. Closures passed as arguments to a procedure must have an environment inferior to the environment in which the call is being made.

The rational for these restrictions will become more apparent in chapter 6 where we describe the runtime system for our implementation.

4.1 Environments as contours

We define environments as pairs of contours. The first contour contains the identifiers and the second contains their associated values. This definition allows us to check environment containment, which is necessary to formalize stack discipline.

- Empty environment

  $\rho_0 = (\langle\,\rangle, \langle\,\rangle)$

- Environment extension

  $(c_i, c_v)[v^*/i^*] = ((i^* \cdot c_i), (v^* \cdot c_v))$

- Environment application

  $$(c_i, c_v)(i) = \begin{cases} \text{if } c_i(i) = \mathit{error} \text{ then } \mathit{error} \\ \text{otherwise } c_v \downarrow (c_i(i)) \end{cases}$$

- Environment equality

$$(\rho =_E \rho') = \begin{cases} \text{if } \rho = \rho_0 \text{ and } \rho' = \rho_0 \text{ then } \mathit{true} \\ \text{if } \rho = \rho''[v^*/i^*] \text{ and } \rho' = \rho'''[v^*/i^*] \text{ then } \rho'' =_E \rho''' \\ \text{otherwise } \mathit{false} \end{cases}$$

- Environment containment - for environments ρ and ρ', if the ρ and ρ' are equal then ρ is contained within ρ'. If they are not equal but ρ' is the extension of an underlying environment ρ'', we check to see if ρ is contained within ρ''. Otherwise ρ is not contained within ρ'.

$$(\rho \subseteq_E \rho') = \begin{cases} \text{if } \rho =_E \rho' \text{ then } \mathit{true} \\ \text{if } \rho \neq_E \rho' \text{ and } \rho' = \rho''[v^*/i^*] \text{ then } \rho \subseteq_E \rho'' \\ \text{otherwise } \mathit{false} \end{cases}$$

- Environment repeal - for variable v, this peels off environment extensions until it finds the outermost extension in which v is defined. It is used to determine if a closure can be assigned to v without violating the upward funarg rule.

$$\rho/v = \begin{cases} \text{if } \rho = \rho_0 \text{ then } \rho \\ \text{if } \rho = \rho'[v^*/i^*] \text{ and } v \in v^* \text{ then } \rho \\ \text{if } \rho = \rho'[w^*/i^*] \text{ and } v \notin w^* \text{ then } \rho'/v \end{cases}$$

4.2 Syntax

4.2.1 Syntactic domains

$\mathtt{K} \in$ Con	constants
$\mathtt{P} \in$ Prim	primitives
$\mathtt{I} \in$ Ide	identifiers
$\mathtt{E} \in$ Exp	expressions

4.2.2 Productions

$$
\begin{array}{lll}
\mathtt{E} & ::= & \mathtt{K} \\
 & | & \mathtt{I} \\
 & | & (\mathtt{set!\ I\ E}) \\
 & | & (\mathtt{if\ E_0\ E_1\ E_2}) \\
 & | & (\mathtt{begin\ E^*\ E}) \\
 & | & (\mathtt{lambda\,(I^*)\ E}) \\
 & | & (\mathtt{P\ E^*}) \\
 & | & (\mathtt{E\ E^*})
\end{array}
$$

4.3 Value domains

$e \in W = \{error\}$	Errors
$B = \{true, false\}$	Booleans
$c \in C$	Characters
$\nu \in N$	Natural Numbers
$\psi \in F = E^* \rightarrow K \rightarrow S \rightarrow A$	Procedures
$\epsilon \in E = B + C + N + F$	Expression Values
$l \in L$	Locations
$\delta \in D = L$	Environment Values
$\rho \in U = \mathrm{Ide}^{*^*} \times D^{*^*}$	Environments
$V = E$	Store Values
$\sigma \in S = D \rightarrow V$	Stores
A	Answers
$\kappa \in K = E \rightarrow S \rightarrow A$	Expression Continuations
$\gamma \in G = E^* \rightarrow S \rightarrow A$	Expression Sequence Continuations

4.4 Semantic valuations

4.4.1 Constants

$\mathcal{K}$ transforms a syntactic constant into an expressible value.

$\mathcal{K} : \text{Con} \rightarrow E$

$\quad \mathcal{K}[\![\texttt{K}]\!] \qquad$ [Implementation Dependent]

4.4.2 Primitives

$\mathcal{P}$ takes a syntactic primitive and returns the associated primitive function. Primitives are discussed further in section 4.6.

$\mathcal{P} : \text{Prim} \rightarrow F$

$\quad \mathcal{P}[\![\texttt{P}]\!] \qquad$ [Implementation Dependent]

4.4.3 Expressions

Given an environment, an expression continuation, and a store, $\mathcal{E}$ evaluates a syntactic expression and returns an answer. Cells allocated from the store are used only for storing the values of variables.

$$\mathcal{E} : \text{Exp} \rightarrow U \rightarrow K \rightarrow S \rightarrow A$$

$$\mathcal{E}[\![\texttt{K}]\!] = \lambda\rho\kappa.\kappa(\mathcal{K}[\![\texttt{K}]\!])$$

$$\mathcal{E}[\![\texttt{I}]\!] = \lambda\rho\kappa.(\rho[\![\texttt{I}]\!] = \mathit{error}) \rightarrow \mathit{wrong} \text{ ``Undefined variable reference''},\ \mathit{deref}\,\rho[\![\texttt{I}]\!]\ \kappa$$

$$\begin{aligned}\mathcal{E}[\![(\texttt{set! I E})]\!] = \lambda\rho\kappa.(\rho[\![\texttt{I}]\!] = \mathit{error}) &\rightarrow \mathit{wrong} \text{ ``Undefined variable reference'' } \sigma, \\ &\mathcal{E}[\![\texttt{E}]\!]\rho(\lambda\epsilon.\,\mathit{funarg\text{-}check}\ \epsilon\ (\rho/[\![\texttt{I}]\!])\ (\mathit{update}\ \rho[\![\texttt{I}]\!]\ \kappa))\end{aligned}$$

$$\mathcal{E}[\![(\texttt{if}\ \texttt{E}_0\ \texttt{E}_1\ \texttt{E}_2)]\!] = \lambda\rho\kappa.\mathcal{E}[\![\texttt{E}_0]\!]\rho(\lambda\epsilon.(\epsilon = \mathit{inB}(\mathit{false})) \rightarrow \mathcal{E}[\![\texttt{E}_2]\!]\rho\kappa,\ \mathcal{E}[\![\texttt{E}_1]\!]\rho\kappa)$$

$$\mathcal{E}[\![(\texttt{begin}\ \texttt{E}^*\ \texttt{E})]\!] = \lambda\rho\kappa.\mathcal{E}^*[\![\texttt{E}]\!]\rho(\lambda\epsilon^*.\mathcal{E}[\![\texttt{E}]\!]\rho\kappa)$$

$$\mathcal{E}[\![(\texttt{lambda}\,(\texttt{I}^*)\ \texttt{E})]\!] = \lambda\rho\kappa.k(\mathit{inF}(\mathit{procedure}\ \texttt{I}^*\ \mathcal{E}[\![\texttt{E}]\!]\ \rho))$$

$$\mathcal{E}[\![(\texttt{P}\ \texttt{E}^*)]\!] = \lambda\rho\kappa.\mathcal{E}^*[\![\texttt{E}^*]\!]\rho(\lambda\epsilon^*.\mathcal{P}[\![\texttt{P}]\!]\ \epsilon^*\ \kappa)$$

$$\mathcal{E}[\![(\texttt{E}\ \texttt{E}^*)]\!] = \lambda\rho\kappa.\mathcal{A}^*[\![\texttt{E}^*]\!]\rho(\lambda\epsilon^*.\mathcal{E}[\![\texttt{E}]\!]\rho(\lambda\epsilon.\ \mathit{apply}\ \epsilon\ \epsilon^*\ (\lambda\epsilon.\,\mathit{funarg\text{-}check}\ \epsilon\ \rho\ \kappa)))$$

4.4.4 Expression sequences

$\mathcal{E}^*$ evaluates a sequence of expressions from left to right and returns the sequence of results to its continuation.

$$\mathcal{E}^* : \text{Exp}^* \to U \to K^* \to S \to A$$
$$\mathcal{E}^*[\![\ ]\!]\rho\gamma = \gamma\langle\,\rangle$$
$$\mathcal{E}^*[\![\texttt{E E}^*]\!]\rho\gamma = \mathcal{E}[\![\texttt{E}]\!]\rho(\lambda\epsilon.\mathcal{E}^*[\![\texttt{E}^*]\!]\rho(\lambda\epsilon^*.k^*(\epsilon :: \epsilon^*)))$$

4.4.5 Argument sequences

$\mathcal{A}^*$ evaluates a sequence of expressions from left to right and returns the sequence of results to its continuation. The value of each expression is checked to assure it does not contain the current environment (unless the current environment is the initial environment).

$$\mathcal{A}^* : \text{Exp}^* \to U \to K^* \to S \to A$$
$$\mathcal{A}^*[\![\ ]\!]\rho\gamma = \gamma\langle\,\rangle$$
$$\mathcal{A}^*[\![\texttt{E E}^*]\!]\rho\gamma = \mathcal{E}[\![\texttt{E}]\!]\rho(\lambda\epsilon.(\rho \neq \rho'[\delta^* /\, \texttt{I}^*]) \to \mathcal{A}^*[\![\texttt{E}^*]\!]\rho(\lambda\epsilon^*.k^*(\epsilon :: \epsilon^*))),$$
$$(\textit{funarg-check}\ \epsilon\,\rho'\,(\lambda\epsilon.\mathcal{A}^*[\![\texttt{E}^*]\!]\rho(\lambda\epsilon^*.k^*(\epsilon :: \epsilon^*))))$$

4.5 Auxiliary functions

alloc tries to allocate n new cells from the store. If there are not enough cells, an error is returned, otherwise a sequence of n locations is returned

$\quad$ $\mathit{alloc} : N \to S \to (W + D^*)$ $\qquad$ [Implementation Dependent]

wrong prints an error message an halts program execution.

$\quad$ $\mathit{wrong} : C^* \to S \to A$ $\qquad$ [Implementation Dependent]

deref applies the store to a store pointer to get the associated cell's value.

$deref: D \rightarrow K \rightarrow S \rightarrow A$

$deref = \lambda\delta\kappa\sigma.\kappa(\sigma\,\delta)\sigma$

funarg-check takes a value and an environment and determines whether the value is an upward funarg for the environment.

$funarg\text{-}check: E \rightarrow U \rightarrow K \rightarrow S \rightarrow A$

$$funarg\text{-}check = \lambda\epsilon\rho\kappa.(\epsilon \neq inF(procedure\ \mathtt{I}^*\ \psi\,\rho')) \rightarrow \kappa\epsilon, \quad (\rho' \not\subseteq_E \rho) \rightarrow wrong \text{ ``Bad environment in closure''}, \kappa\epsilon$$

update modifies a cell in the store so that it contains a new value.

$update: D \rightarrow K \rightarrow E \rightarrow S \rightarrow A$

$update = \lambda\delta\kappa\epsilon\sigma.\kappa\epsilon(\sigma[\epsilon/\delta])$

procedure creates a closure.

$procedure: \text{Ide}^* \rightarrow (U \rightarrow K \rightarrow S \rightarrow A) \rightarrow U \rightarrow F$

$$\begin{aligned} procedure = \lambda\,\mathtt{I}^*\,\psi\rho.(\lambda\epsilon^*\kappa\sigma.(\#\epsilon^* \neq \#\,\mathtt{I}^*) &\rightarrow wrong \text{ ``Wrong number of arguments''}\ \sigma, \\ &\text{case } (alloc\ \#\,\mathtt{I}^*\ \sigma) \\ &\quad \lambda e.\ wrong \text{ ``Out of memory''}\ \sigma \\ &\quad \lambda\delta^*.\psi(\rho[\delta^*/\,\mathtt{I}^*])\kappa(\sigma[\epsilon^*/\delta^*])) \end{aligned}$$

apply applies a closure to its arguments.

$apply: E \rightarrow E^* \rightarrow K \rightarrow S \rightarrow A$

$apply = \lambda\epsilon\epsilon^*\kappa.(\epsilon \neq inF(\psi)) \rightarrow wrong \text{ ``Non-procedure to apply''}, \psi\epsilon^*\kappa$

4.6 Primitives

Primitives are functions which are built into the language and are accessed through the $\mathcal{P}$ valuation. We view primitives as interfaces to underlying machine operations (such as complement, add, etc.). Rather than specify a canonical set of machine operations, we instead abstract their behavior. Each operator takes expressible values as arguments and returns either an expressible value or an error code as its result. Hence we envision a set of unary operators with type:

$$unop_i : E \rightarrow (E + C^*)$$

and similarly a set of binary operators with type:

$$binop_i : E \rightarrow E \rightarrow (E + C^*)$$

Each operator has an associated primitive function. When invoked, the primitive will verify that it has been given the proper number of arguments, pass the arguments to the operator, and then check the return value. If the value is an error code, the primitive halts execution otherwise it returns the value to its continuation.

$$
\begin{aligned}
&unprim_i : F \\
&unprim_i = \lambda\epsilon^*\kappa.(\epsilon^* \neq \langle\epsilon\rangle) \rightarrow wrong \text{ "Wrong number of arguments"}, \\
&\qquad\qquad \text{case } (unop_i\ \epsilon) \\
&\qquad\qquad\quad \lambda\epsilon.\kappa\epsilon \\
&\qquad\qquad\quad \lambda c^*.\ wrong\ c^*
\end{aligned}
$$

$$
\begin{aligned}
&binprim_i : F \\
&binprim_i = \lambda\epsilon^*\kappa.(\epsilon^* \neq \langle\epsilon, \epsilon'\rangle) \rightarrow wrong \text{ "Wrong number of arguments"}, \\
&\qquad\qquad \text{case } (binop_i\ \epsilon\,\epsilon') \\
&\qquad\qquad\quad \lambda\epsilon.\kappa\epsilon \\
&\qquad\qquad\quad \lambda c^*.\ wrong\ c^*
\end{aligned}
$$

There are few restrictions on the behavior of primitives but they must enforce the "no upward funargs" rule. Hence if an implementation adds a heap and some data structure primitives, the implementation must ensure that no upward funargs are ever returned in the data structures. Many stack discipline languages accomplish this simply by not allowing closures to be put into any data structures, but we leave that up to the implementor's discretion.

```
(letrec
    ((fact (lambda (i a) (if (zero? i) a (fact (- i 1) (* a i))))))
  (fact 4 1))
```

Figure 5: Iterative Factorial

```
((lambda (fact)
    (begin
      (set! fact (lambda (i a) (if (zero? i) a (fact (- i 1) (* a i)))))
      (fact 4 1)))
 0)
```

Figure 6: Iterative Factorial (with *letrec* expanded)

4.7 Factorial example

In Figure 5 we present the standard iterative form of the factorial function (Figure 5 uses a *letrec* for readablity, Figure 6 shows the result of expanding the *letrec*). The function takes two arguments: i - the value for which we wish to calculate factorial, and a - an accumulator. If i is 0 we return the value we have accumulated, otherwise we recur by decrementing i and multiplying the accumulator by i. The function is initially called with the value we wish to calculate factorial for (4 in the example) and the accumulator set to 1.

Chapter 5

The Compiler and Abstract Machine

In this chapter we present and verify a compiler for ν-LISP based on the techniques of [12, 71]. Our goals in this chapter are to:

- define a compiler for ν-LISP
- define an abstract machine for executing the code generated by our compiler
- verify the compiler and abstract machine with respect to the original language definition
- provide an operational definition of the abstract machine

5.1 An informal overview

We will use the techniques described in [12, 71, 45] to derive our compiler and abstract machine. These techniques require the compiler writer to extract the static and dynamic components of the language from its formal definition. A compiler is then written using the static components to produce abstract machine code. The instructions are defined to manipulate the dynamic components in such a way that the static and dynamic components compose to form a language definition equivalent to the original one. In the case of ν-LISP the compiler works on two static components:

- τ - a symbol table which is a contour consisting of identifiers. It is used to turn identifiers into lexical addresses at compile time.
- π - a sequel which is the code to follow the code for the current expression. This is the statically known portion of the continuation.

The compiler generates a finite number of well defined instructions which manipulate the dynamic components of the language. The instruction definitions can be considered to comprise a definition for an abstract machine which manipulates the dynamic language components. For ν-LISP, there are four dynamic components:

- μ - a runtime environment which is a contour used to transform lexical addresses into denoted values (pointers into the variable store).
- ζ - a local stack used for storing intermediate results. These correspond to temporary variables traditional intermediate representations produce.
- κ - a runtime continuation. This is an activation record used to implement procedure calls and returns.
- σ - the variable store. This is the same as in the original semantics.

To execute a phrase in the source language, it is first compiled using a symbol table and a code sequel. The code produced by the compiler can then be executed on the abstract machine.

Each compiler valuation has an associated correctness criterion relating the compiler semantics to the original. For example, the criterion for compiling expressions is:

$$(\mathcal{CE}[\![\mathtt{E}]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)\sigma$$

Intuitively, this says that executing the code for expression E is the same as evaluating the expression. Both result in the execution of the code sequel π with the value of the expression pushed onto the runtime stack. This also gives a clear indication of how the original valuation parameters have been split across the static and dynamic borders.

The two contours that comprise the environment have been split into a static symbol table and runtime display. These contours pair off to perform the same function as the original environment.

The continuation also has its static and dynamic components. The code sequel is the static part of the continuation. Intuitively, this means that at compile time it is known what code will follow the code for a particular expression. There dynamic components of the continuation are the dynamic state necessary to execute the sequel.

The other compiler valuations work in a similar manner. We summarize the valuations and their correctness criteria here:

- $\mathcal{CP}$ is used to compile primitives. Its correctness criterion relates that primitives pop their arguments from the top of the stack and push their result.

  $(\mathcal{CP}[\![\mathtt{P}]\!]\nu\pi)\mu\zeta\kappa\sigma = \mathcal{P}[\![\mathtt{P}]\!]\langle(\zeta \downarrow (\nu - 1)), \ldots, (\zeta \downarrow 0)\rangle(\lambda\epsilon.\pi\mu(\epsilon :: (\zeta \dagger \nu))\kappa)\sigma$

- $\mathcal{CT}$ is similar to $\mathcal{CE}$ except that it compiles expressions in tail position. Expressions in tail position evaluate using the current continuation rather than constructing a new one.

  $(\mathcal{CT}[\![\mathtt{E}]\!]\tau)\mu\langle\,\rangle\kappa\sigma = \mathcal{E}[\![\mathtt{E}]\!](\tau, \mu)\kappa\sigma$

- $\mathcal{CE}^*$ is used to compile a sequence of expressions. At runtime, each expression pushes its value on the stack. Due to the left to right order of evaluation, the values for the expressions end up on the stack in reverse order.

  $(\mathcal{CE}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}^*[\![\mathtt{E}^*]\!](\tau, \mu)(\lambda\epsilon^*.\pi\mu((\rightleftharpoons \epsilon^*) \S \zeta)\kappa)\sigma$

- $\mathcal{CA}^*$ compiles code in a manner similar to $\mathcal{CE}^*$ except that its expressions are arguments to procedures. At runtime, the value produced by each expression is checked to ensure that it is a legal parameter to a procedure.

$$(\mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{A}^*[\![\mathtt{E}^*]\!](\tau,\mu)(\lambda\epsilon^*.\pi\mu((\rightleftharpoons \epsilon^*)\,\S\,\zeta)\kappa)\sigma$$

- $\mathcal{CS}^*$ is also similar to $\mathcal{CE}^*$ except that the code it generates does not save the values of the expressions on the stack at runtime.

$$(\mathcal{CS}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}^*[\![\mathtt{E}^*]\!](\tau,\mu)(\lambda\epsilon^*.\pi\mu\zeta\kappa)\sigma$$

The rest of the chapter is dedicated to formalizing the compiler definition, deriving the actions of the abstract machine, and verifying that compiler and abstract machine satisfy the correctness criteria.

5.2 The compiler

5.2.1 Compiler domains

$\iota \in I = (N \times N)$	deBruijn Indices (lexical addresses)
$\tau \in U_c = \mathrm{Ide}^{**}$	Symbol Tables
$\mu \in U_r = D^{**}$	Run-Time Environments (displays)
$\zeta \in Z = E^*$	Local Stacks
$\pi \in P = U_r \to Z \to K \to S \to A$	Programs

5.2.2 Symbol tables and displays

As specified in the language definition, ν-LISP environments are pairs of contours. The first contour is used to map identifiers to lexical addresses while the second is used to map lexical addresses to values. In the compiler, we split an environment across the static and dynamic line.

On the static side we have a symbol table, τ, which corresponds to the first component of an environment. During compilation, the symbol table transforms references to identifiers into lexical addresses. At runtime the second component on the environment, μ, transforms these lexical addresses into values.

To extend an environment by a sequence of identifiers and values, we extend each of the components of the environment:

$$(\tau, \mu)[\delta^* / \mathtt{I}^*] = ((\mathtt{I}^* \cdot \tau), (\delta^* \cdot \mu))$$

Hence the symbol table can be extended at compile time by the sequence of identifiers. At runtime, when the values are available, the display can then be extended with them.

Symbol tables and the runtime displays compose to form environments. If a symbol table knows about an identifier at compile time, the associated display should know about the identifier's lexical address at runtime otherwise the environment is invalid. We formally define this property by giving a relationship between symbol tables and displays.

Definition 1 *We say symbol table τ and runtime display μ are* reasonable *if $\tau \approx \mu$.*

Lemma 1 *If τ and μ are reasonable then $\tau(\mathtt{I}) = \iota \iff (\mu \downarrow \iota) \neq error$*

Proof: Structural induction on contours.

In the language definition, the check for upward funargs is done by by various containment checks on the environment. In the case of assignment, the environment has to be "repealed" to the defining contour to assure that the value being assigned cannot be an upward funarg. This check now has to be performed at runtime, so we define an analogous repeal operation for displays based on lexical addresses:

$$\mu/(d,o) = \begin{cases} \text{if } d = 0 \text{ then } \mu \\ \text{if } \mu = (t^* \cdot \mu') \text{ and } d = d' + 1 \text{ then } \mu'/(d', o) \end{cases}$$

5.2.3 Compiler valuations

We now formally define the compiler valuations described in section 5.1.

Compile primitive

Compiling primitives is implementation dependent, as is the primitive valuation. Regardless of how it works, $\mathcal{CP}$ takes as input the number of arguments and the code sequel and is required to produce the code that will pop the arguments off the stack, push the result, and execute the sequel.

$\mathcal{CP} : \mathrm{Prim} \to N \to P \to P$

$\quad \mathcal{CP}[\![\mathtt{P}]\!] \qquad$ [Implementation Dependent]

Compile expression

Given an expression, a symbol table, and a sequel, $\mathcal{CE}$ produces the code for the expression followed by the sequel. At runtime, execution of the generated code pushes the value of the expression on top of the local stack, and then executes the sequel.

For example, to generate code for a constant, we generate an instruction that will push the constant on the stack at runtime and then follow the instruction by the code sequel. Generating code for the other phrases in the language follows a similar format. There are a few subtleties that occur in the code generator which we summarize:

- The true and false branches of the conditional phrase are compiled using the same sequel. At this point our program tree is actually a dag. To aid in linearizing the code later, we insert *goto* instructions to identify these points.
- The body of a procedure is a tail expression so we compile it using the $\mathcal{CT}$ valuation.
- When generating the code for a procedure application, we first emit a *catch* instruction that will create a continuation at runtime with the sequel as the return point.

$\mathcal{CE} : \mathrm{Exp} \to U_c \to P \to P$

$\quad \mathcal{CE}[\![\mathtt{K}]\!]\tau\pi = (\mathit{constant}\ \mathcal{K}[\![\mathtt{K}]\!]\ \pi)$

$\quad \mathcal{CE}[\![\mathtt{I}]\!]\tau\pi = (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \to \mathit{fetch\text{-}error}, (\mathit{fetch}\ \tau([\![\mathtt{I}]\!])\ \pi)$

$$\mathcal{CE}[\![(\texttt{set! I E})]\!]\tau\pi = (\tau([\![\texttt{I}]\!]) = \mathit{error}) \rightarrow \mathit{assign\text{-}error},\ \mathcal{CE}[\![\texttt{E}]\!]\tau(\mathit{assign}\ \tau([\![\texttt{I}]\!])\ \pi)$$

$$\mathcal{CE}[\![(\texttt{if}\ \texttt{E}_0\ \texttt{E}_1\ \texttt{E}_2)]\!]\tau\pi = \mathcal{CE}[\![\texttt{E}_0]\!]\tau(\mathit{brf}\,(\mathcal{CE}[\![\texttt{E}_1]\!]\tau(\mathit{goto}\ \pi))\ (\mathcal{CE}[\![\texttt{E}_2]\!]\tau(\mathit{goto}\ \pi)))$$

$$\mathcal{CE}[\![(\texttt{begin}\ \texttt{E}^*\ \texttt{E})]\!]\tau\pi = \mathcal{CS}^*[\![\texttt{E}^*]\!]\tau(\mathcal{CE}[\![\texttt{E}]\!]\tau\pi)$$

$$\mathcal{CE}[\![(\texttt{lambda}\,(\texttt{I}^*)\ \texttt{E})]\!]\tau\pi = (\mathit{close}\ \#\,\texttt{I}^*\ (\mathcal{CT}[\![\texttt{E}]\!]((\Leftarrow\ \texttt{I}^*)\cdot\tau))\ \pi)$$

$$\mathcal{CE}[\![(\texttt{P}\ \texttt{E}^*)]\!]\tau\pi = \mathcal{CE}^*[\![\texttt{E}^*]\!]\tau(\mathcal{CP}[\![\texttt{P}]\!]\ \#\,\texttt{E}^*\ \pi)$$

$$\mathcal{CE}[\![(\texttt{E}\ \texttt{E}^*)]\!]\tau\pi = (\mathit{catch}\ \pi\ (\mathcal{CA}^*[\![\texttt{E}^*]\!]\tau(\mathcal{CE}[\![\texttt{E}]\!]\tau(\mathit{jsr}\ \#\,\texttt{E}^*))))$$

Compile tail expression

$\mathcal{CT}$ compiles expressions in tail position. It works much the same way $\mathcal{CE}$ does except that it needs no static continuation and procedure calls do not create a new continuation at runtime (the current continuation is used).

Since there is no code continuation to $\mathcal{CT}$, the code generated for branches do not form dags in the program graph. To aid in code linearization, we distinguish the branch instruction generated by $\mathcal{CT}$ from the one generated by $\mathcal{CE}$ (i.e. *brfng* rather than *brf*).

A subtle issue arises when compiling procedure calls. Our language has been designed such that the arguments to procedure calls are guaranteed not to have references to the topmost environment extension. If the procedure being called also does not refer to the topmost extension, than this extension is no longer visible anywhere (and can be garbage collected). We distinguish calls to these kinds of procedures using a *jmp* instruction rather than a *jsr* instruction. The semantic meaning of these two instructions is the same, however they will be implemented quite differently[1].

$$\mathcal{CT} : \mathrm{Exp} \rightarrow U_c \rightarrow P$$

$$\mathcal{CT}[\![\texttt{K}]\!]\tau = (\mathit{constant}\ \mathcal{K}[\![\texttt{K}]\!]\ \mathit{rts})$$

$$\mathcal{CT}[\![\texttt{I}]\!]\tau = (\tau([\![\texttt{I}]\!]) = \mathit{error}) \rightarrow \mathit{fetch\text{-}error}, (\mathit{fetch}\ \tau([\![\texttt{I}]\!])\ \mathit{rts})$$

$$\mathcal{CT}[\![(\texttt{set! I E})]\!]\tau = (\tau([\![\texttt{I}]\!]) = \mathit{error}) \rightarrow \mathit{assign\text{-}error},$$

[1] We identify procedures that can be called with a *jmp* by checking if the application is an identifier that is not defined in the topmost environment extension. More extensive analysis could be incorporated to identify other instances when *jmp*'s could be used but this will suffice for our purposes.

$$\mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathit{assign}\ \tau([\![\mathtt{I}]\!])\ \mathit{rts})$$

$$\mathcal{CT}[\![(\mathtt{if}\ \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!]\tau = \mathcal{CE}[\![\mathtt{E}_0]\!]\tau(\mathit{brfng}(\mathcal{CT}[\![\mathtt{E}_1]\!]\tau)\,(\mathcal{CT}[\![\mathtt{E}_2]\!]\tau))$$

$$\mathcal{CT}[\![(\mathtt{begin}\ \mathtt{E}^*\ \mathtt{E})]\!]\tau = \mathcal{CS}^*[\![\mathtt{E}^*]\!]\tau(\mathcal{CT}[\![\mathtt{E}]\!]\tau)$$

$$\mathcal{CT}[\![(\mathtt{lambda}\,(\mathtt{I}^*)\ \mathtt{E})]\!]\tau = (\mathit{close}\ \#\,\mathtt{I}^*\,(\mathcal{CT}[\![\mathtt{E}]\!]((\Leftarrow\ \mathtt{I}^*)\cdot\tau)\,\mathit{rts})\ \mathit{rts})$$

$$\mathcal{CT}[\![(\mathtt{P}\ \mathtt{E}^*)]\!]\tau = \mathcal{CE}^*[\![\mathtt{E}^*]\!]\tau(\mathcal{CP}[\![\mathtt{P}]\!]\ \#\,\mathtt{E}^*\ \mathit{rts})$$

$$\begin{aligned}\mathcal{CT}[\![(\mathtt{I}\ \mathtt{E}^*)]\!]\tau = &(\tau([\![\mathtt{I}]\!]) = (0,o)) \to \mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau(\mathcal{CE}[\![\mathtt{I}]\!]\tau(\mathit{jsr}\ \#\,\mathtt{E}^*)),\\ &\mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau(\mathcal{CE}[\![\mathtt{I}]\!]\tau(\mathit{jmp}\ \#\,\mathtt{E}^*))\end{aligned}$$

$$\mathcal{CT}[\![(\mathtt{E}\ \mathtt{E}^*)]\!]\tau = \mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau(\mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathit{jsr}\ \#\,\mathtt{E}^*))$$

Compile expression sequences

$\mathcal{CE}^*$ compiles a sequence of expressions. At runtime, the values of the expressions are saved by pushing them on the runtime stack. The expressions are evaluated from left to right. As each expression is evaluated, its value gets pushed onto the stack and the next expression evaluates. Hence, the values end up on the stack in reverse order.

$$\mathcal{CE}^* : \mathrm{Exp} \to U_c \to P \to P$$

$$\mathcal{CE}^*[\![\ ]\!]\tau\pi = \pi$$

$$\mathcal{CE}^*[\![\mathtt{E}\ \mathtt{E}^*]\!]\tau\pi = \mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathcal{CE}^*[\![\mathtt{E}^*]\!]\tau\pi)$$

Compile argument sequences

$\mathcal{CA}^*$ compiles a sequence of arguments. At runtime, the value of each argument is checked to ensure that it is a legal parameter to a procedure. Due to the left to right order of evaluation, parameters end up on the stack in reverse order.

$$\mathcal{CA}^* : \mathrm{Exp} \to U_c \to P \to P$$

$$\mathcal{CA}^*[\![\ ]\!]\tau\pi = \pi$$

$$\mathcal{CA}^*[\![\mathtt{E}\ \mathtt{E}^*]\!]\tau\pi = \mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathit{test\text{-}arg}\ \mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau\pi))$$

Compile Statements

$\mathcal{CS}^*$ compiles a sequence of expressions in a manner similar to $\mathcal{CE}^*$. The difference is that the values produced at runtime are discarded rather than saved on the stack.

$$\mathcal{CS}^* : \text{Exp} \rightarrow U_c \rightarrow P \rightarrow P$$
$$\mathcal{CS}^*[\![\ ]\!]\tau\pi = \pi$$
$$\mathcal{CS}^*[\![\texttt{E E}^*]\!]\tau\pi = \mathcal{CE}[\![\texttt{E}]\!]\tau(pop(\mathcal{CS}^*[\![\texttt{E}^*]\!]\tau\pi))$$

5.3 The abstract machine

5.3.1 Procedure definitions

In the language definition, there are two flavors of procedures created - continuations and closures. The abstract machine definition of these procedures is specified here.

Continuations

$$(\texttt{cont}\ \pi\mu\zeta\kappa) = (\lambda\epsilon.\ return\ \epsilon\,(\tau,\mu)\,(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa))$$

Closures

$$(\texttt{closure}\ \#\ \texttt{I}^*\ \mu\,(\mathcal{CT}[\![\texttt{E}]\!](\texttt{I}^* \cdot \tau))) = inF(procedure\ \texttt{I}^*\ \mathcal{E}[\![\texttt{E}]\!]\,(\tau,\mu))$$

5.3.2 Machine instructions

The machine instructions form the code produced by the compiler. Their actions define the behavior of the abstract machine and must manipulate the runtime structures in a manner consistent with the correctness criteria.

$$constant : E \rightarrow P \rightarrow P$$
$$(constant\ \epsilon\,\pi) = \lambda\mu\zeta\kappa\sigma.(\pi\mu(\epsilon :: \zeta)\kappa\sigma)$$

$$fetch\text{-}error : P$$
$$fetch\text{-}error = \lambda\mu\zeta\kappa\sigma.\ wrong\ \text{"Undefined variable reference."}\ \sigma$$

$$fetch : I \rightarrow P \rightarrow P$$
$$(fetch\ \iota\, \pi) = \lambda\mu\zeta\kappa\sigma.(\pi\mu((\sigma\,(\mu \downarrow \iota) \mid E) :: \zeta)\kappa\sigma)$$

$$assign\text{-}error : P$$
$$assign\text{-}error = \lambda\mu\zeta\kappa\sigma.\ wrong\ \text{“Undefined variable to update.”}\ \sigma$$

$$assign : I \rightarrow P \rightarrow P$$
$$\begin{aligned}(assign\ \iota\, \pi) = \lambda\mu\zeta\kappa\sigma.&(\zeta \neq \epsilon :: \zeta') \rightarrow wrong\ \text{“No value to assign.”}\ \sigma,\\ &(\epsilon \neq (\texttt{closure}\ \nu\, \mu'\, \pi')) \rightarrow (\pi\mu\zeta\kappa\sigma[\epsilon/(\mu \downarrow \iota)]),\\ &(\mu' \not\subseteq_C (\mu/\iota)) \rightarrow wrong\ \text{“Bad environment in closure”}\ \sigma,\\ &(\pi\mu\zeta\kappa\sigma[\epsilon/(\mu \downarrow \iota)])\end{aligned}$$

$$brf : P \rightarrow P \rightarrow P$$
$$\begin{aligned}(brf\ \pi_0\, \pi_1) = \lambda\mu\zeta\kappa\sigma.&(\zeta \neq \epsilon :: \zeta') \rightarrow wrong\ \text{“No branch value”}\ \sigma,\\ &(\epsilon = inB(false)) \rightarrow (\pi_1\mu\zeta'\kappa\sigma), (\pi_0\mu\zeta'\kappa\sigma))\end{aligned}$$

$$brfng : P \rightarrow P \rightarrow P$$
$$\begin{aligned}(brfng\ \pi_0\, \pi_1) = \lambda\mu\zeta\kappa\sigma.&(\zeta \neq \epsilon :: \zeta') \rightarrow wrong\ \text{“No branch value”}\ \sigma,\\ &(\epsilon = inB(false)) \rightarrow (\pi_1\mu\zeta'\kappa\sigma), (\pi_0\mu\zeta'\kappa\sigma))\end{aligned}$$

$$goto : P \rightarrow P$$
$$(goto\ \pi) = \lambda\mu\zeta\kappa\sigma.(\pi\mu\zeta\kappa\sigma)$$

$$close : N \rightarrow P \rightarrow P \rightarrow P$$
$$(close\ \nu\, \pi_0\, \pi_1) = \lambda\mu\zeta\kappa\sigma.(\pi_1\mu((\texttt{closure}\ \nu\, \mu\, \pi_0) :: \zeta)\kappa\sigma)$$

$$rts : P$$
$$\begin{aligned}(rts) = \lambda\mu\zeta\kappa\sigma.&(\zeta \neq (\epsilon :: \zeta')) \rightarrow wrong\ \text{“No value to return”}\ \sigma,\\ &\kappa\epsilon\sigma\end{aligned}$$

$$catch : P \rightarrow P \rightarrow P$$
$$(catch\ \pi_0\, \pi_1) = \lambda\mu\zeta\kappa\sigma.(\pi_1\mu\langle\,\rangle(\texttt{cont}\ \pi_0\mu\zeta\kappa)\sigma)$$

$jmp : N \rightarrow P$

$(jmp\ \nu) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq (inF(\psi) :: \zeta')) \rightarrow wrong$ "Non-procedure to apply" $\sigma,$
$(\psi\zeta'\kappa\sigma)$

$jsr : N \rightarrow P$

$(jsr\ \nu) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq (inF(\psi) :: \zeta')) \rightarrow wrong$ "Non-procedure to apply" $\sigma,$
$(\psi\zeta'\kappa\sigma)$

$test\text{-}arg : P \rightarrow P$

$(test\text{-}arg\ \pi) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq \epsilon :: \zeta') \rightarrow wrong$ "No value to test." $\sigma,$
$(\epsilon \neq (\texttt{closure}\ \nu\ \mu'\ \pi')) \rightarrow (\pi\mu\zeta\kappa\sigma),$
$(\mu' \not\subset_C (\mu/\iota)) \rightarrow wrong$ "Bad environment in closure" $\sigma,$
$(\pi\mu\zeta\kappa\sigma)$

$pop : P \rightarrow P$

$(pop\ \pi) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq \epsilon :: \zeta') \rightarrow wrong$ "Empty stack to pop." $\sigma, (\pi\mu\zeta'\kappa\sigma)$

5.3.3 Compiler primitives

The compiler primitives are also machine instructions. This approach to primitives corresponds to the notion of primitives as simple, inlined functions. As with the original semantics, primitives are defined in terms of underlying machine operations.

$sm\text{-}unprim_i : P \rightarrow P$

$(sm\text{-}unprim_i\ \pi) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq \epsilon :: \zeta') \rightarrow wrong$ "Wrong number of arguments" $\sigma,$
case $unop_i\ \epsilon$
$\lambda\epsilon.\pi\mu(\epsilon :: \zeta')\kappa\sigma$
$\lambda c^*.\ wrong\ c^*$

$sm\text{-}binprim_i : P \rightarrow P$

$(sm\text{-}binprim_i\ \pi) = \lambda\mu\zeta\kappa\sigma.(\zeta \neq \epsilon' :: \epsilon :: \zeta') \rightarrow wrong$ "Wrong number of arguments" $\sigma,$

case $binop_i\, \epsilon\epsilon'$

$\lambda\epsilon.\pi\mu(\epsilon :: \zeta')\kappa\sigma$

$\lambda c^*.\, wrong\ c^*$

5.4 Correctness of the compiler and abstract machine

We now set out to verify that the compiler and abstract machine satisfy the correctness criteria described in section 5.1. The proof consists of 6 simultaneous inductions, all of which assume that the symbol tables and displays are reasonable. Rather than subject the reader to the entire proof, we instead present some of the more interesting cases[2].

The following lemma is useful when relating the checks for upward funargs. It states that if one runtime display is contained within another, then the analogous containment is also true of the environments derived from the displays and their corresponding symbol tables.

Lemma 2 *If μ and τ are reasonable, μ' and τ' are reasonable, and $\mu' \subseteq_C \mu$ then*

$$(\tau', \mu') \subseteq_E (\tau, \mu)$$

Proof: Structural induction on environments.

5.4.1 Primitives

$$(\mathcal{CP}[\![\mathtt{P}]\!]\nu\pi)\mu\zeta\kappa\sigma = \mathcal{P}[\![\mathtt{P}]\!]\langle(\zeta \downarrow (\nu - 1)), \ldots, (\zeta \downarrow 0)\rangle(\lambda\epsilon\sigma.\pi\mu(\zeta \dagger \nu)\kappa\sigma)\sigma$$

Proof: [Implementation Dependent]

5.4.2 Expressions

$$(\mathcal{CE}[\![\mathtt{E}]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}[\![\mathtt{E}]\!](\tau, \mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)\sigma$$

[2]The interested reader is referred to [45] for a similar proof shown in its entirety.

Proof: Structural induction on expressions. We show a few of the interesting cases. Unless otherwise noted, each step in the following proofs is either a straightforward application of the induction hypotheses or by definition. In these steps, we underline the portion of the proof being transformed.

$$
\begin{aligned}
&(\mathcal{CE}[\![\mathtt{K}]\!]\tau\pi)\mu\zeta\kappa\sigma \\
&= (\underline{constant}\,\mathcal{K}[\![\mathtt{K}]\!]\ \pi)\mu\zeta\kappa\sigma \\
&= (\pi\mu(\mathcal{K}[\![\mathtt{K}]\!] :: \zeta)\kappa\sigma) \\
&= \mathcal{E}[\![\mathtt{K}]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)\sigma
\end{aligned}
$$

$$
\begin{aligned}
&(\mathcal{CE}[\![(\mathtt{if}\ \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!]\tau\pi)\mu\zeta\kappa\sigma \\
&= (\mathcal{CE}[\![\mathtt{E}_0]\!]\tau(\mathit{brf}\,(\mathcal{CE}[\![\mathtt{E}_1]\!]\tau(\underline{goto}\,\pi))\ (\mathcal{CE}[\![\mathtt{E}_2]\!]\tau(\underline{goto}\,\pi))))\mu\zeta\kappa\sigma \\
&= \underline{(\mathcal{CE}[\![\mathtt{E}_0]\!]\tau(\mathit{brf}\,(\mathcal{CE}[\![\mathtt{E}_1]\!]\tau\pi)\ (\mathcal{CE}[\![\mathtt{E}_2]\!]\tau\pi)))\mu\zeta\kappa\sigma} \\
&= \mathcal{E}[\![\mathtt{E}_0]\!](\tau,\mu)(\lambda\epsilon.(\underline{brf}\,(\mathcal{CE}[\![\mathtt{E}_1]\!]\tau\pi)\ (\mathcal{CE}[\![\mathtt{E}_2]\!]\tau\pi))\mu(\epsilon :: \zeta)\kappa)\sigma \\
&= \mathcal{E}[\![\mathtt{E}_0]\!](\tau,\mu)(\lambda\epsilon.(\epsilon = \mathit{inB}(\mathit{false})) \rightarrow \underline{(\mathcal{CE}[\![\mathtt{E}_2]\!]\tau\pi)\mu\zeta\kappa}, \\
&\qquad\qquad \underline{(\mathcal{CE}[\![\mathtt{E}_1]\!]\tau\pi)\mu\zeta\kappa})\sigma \\
&= \mathcal{E}[\![\mathtt{E}_0]\!](\tau,\mu)(\lambda\epsilon.(\epsilon = \mathit{inB}(\mathit{false})) \rightarrow \mathcal{E}[\![\mathtt{E}_2]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa), \\
&\qquad\qquad \mathcal{E}[\![\mathtt{E}_1]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa))\sigma \\
&= \mathcal{E}[\![(\mathtt{if}\ \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)\sigma
\end{aligned}
$$

$$
\begin{aligned}
&(\mathcal{CE}[\![(\mathtt{set!}\ \mathtt{I}\ \mathtt{E})]\!]\tau\pi)\mu\zeta\kappa\sigma \\
&= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \rightarrow \underline{\mathit{assign\text{-}error}}\,\mu\zeta\kappa\sigma, \\
&\quad (\mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathit{assign}\ \tau([\![\mathtt{I}]\!])\,\pi))\mu\zeta\kappa\sigma \\
&= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \rightarrow \mathit{wrong}\ \text{“Undefined variable to update.”}\ \sigma, \\
&\quad \underline{(\mathcal{CE}[\![\mathtt{E}]\!]\tau(\mathit{assign}\ \tau([\![\mathtt{I}]\!])\ \pi))\mu\zeta\kappa\sigma} \\
&= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \rightarrow \mathit{wrong}\ \text{“Undefined variable to update.”}\ \sigma, \\
&\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.(\underline{assign}\ \tau([\![\mathtt{I}]\!])\ \pi)\mu(\epsilon :: \zeta)\kappa)\sigma \\
&= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \rightarrow \mathit{wrong}\ \text{“Undefined variable to update.”}\ \sigma, \\
&\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.(\epsilon \neq (\mathtt{closure}\ \#\ \mathtt{I}^*\ \mu'\ (\mathcal{CT}[\![\mathtt{E}]\!](\mathtt{I}^* \cdot \tau')))) \rightarrow \underline{\lambda\sigma.(\pi\mu(\epsilon :: \zeta)\kappa\sigma[\epsilon/(\mu \downarrow \tau([\![\mathtt{I}]\!]))])}, \\
&\qquad\qquad (\mu' \not\subseteq_C (\mu/\tau([\![\mathtt{I}]\!]))) \rightarrow \mathit{wrong}\ \text{“Bad environment in closure”},
\end{aligned}
$$

$$\underline{\lambda\sigma.(\pi\mu(\epsilon :: \zeta)\kappa\sigma[\epsilon/(\mu \downarrow \tau([\![\mathtt{I}]\!]))])}\sigma$$

$= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \to \mathit{wrong}$ “Undefined variable to update.” σ,

$\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.(\epsilon \neq \underline{(\mathtt{closure}\ \#\ \mathtt{I}^*\ \mu'\ (\mathcal{CT}[\![\mathtt{E}]\!](\mathtt{I}^* \cdot \tau')))}) \to (\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)),$

$\qquad (\mu' \not\subseteq_C (\mu/\tau([\![\mathtt{I}]\!]))) \to \mathit{wrong}$ “Bad environment in closure”,

$\qquad (\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)))\sigma$

$= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \to \mathit{wrong}$ “Undefined variable to update.” σ,

$\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.(\epsilon \neq (\mathit{procedure}\ \mathtt{I}^*\ \mathcal{E}[\![\mathtt{E}]\!]\ (\tau',\mu'))) \to (\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)),$

$\qquad \underline{(\mu' \not\subseteq_C (\mu/\tau([\![\mathtt{I}]\!])))} \to \mathit{wrong}$ “Bad environment in closure”,

$\qquad (\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)))\sigma$

by Lemma 2

$= (\tau([\![\mathtt{I}]\!]) = \mathit{error}) \to \mathit{wrong}$ “Undefined variable to update.” σ,

$\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.\underline{(\epsilon \neq (\mathit{procedure}\ \mathtt{I}^*\ \mathcal{E}[\![\mathtt{E}]\!]\ (\tau',\mu'))) \to (\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)),}$

$\qquad \underline{((\tau',\mu') \not\subseteq_C ((\tau,\mu)/[\![\mathtt{I}]\!])) \to \mathit{wrong}\ \text{“Bad environment in closure”},}$

$\qquad \underline{(\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa))})\sigma$

$= (\underline{\tau([\![\mathtt{I}]\!])} = \mathit{error}) \to \mathit{wrong}$ “Undefined variable to update.” σ,

$\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.\,\mathit{funarg\text{-}check}\ \epsilon\,(\tau,\mu)\,(\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)))\sigma$

since τ and μ are reasonable

$= ((\tau,\mu)[\![\mathtt{I}]\!] = \mathit{error}) \to \mathit{wrong}$ “Undefined variable to update.” σ,

$\quad \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)(\lambda\epsilon.\,\mathit{funarg\text{-}check}\ \epsilon\,(\tau,\mu)\,(\mathit{update}\ (\tau,\mu)[\![\mathtt{I}]\!]\ (\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)))\sigma$

$= \mathcal{E}[\![(\mathtt{set!\ I\ E})]\!](\tau,\mu)(\lambda\epsilon.\pi\mu(\epsilon :: \zeta)\kappa)\sigma$

5.4.3 Tail expressions

$$(\mathcal{CT}[\![\mathtt{E}]\!]\tau)\mu\langle\,\rangle\kappa\sigma = \mathcal{E}[\![\mathtt{E}]\!](\tau,\mu)\kappa\sigma$$

Proof: Similar to $\mathcal{CE}$.

5.4.4 Expression sequences

$$(\mathcal{CE}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}^*[\![\mathtt{E}^*]\!](\tau,\mu)(\lambda\epsilon^*.\pi\mu((\rightleftharpoons \epsilon*)\ \S\ \zeta)\kappa)\sigma$$

Proof: Induction on the length of the expression sequences.

5.4.5 Argument sequences

$$(\mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{A}^*[\![\mathtt{E}^*]\!](\tau,\mu)(\lambda\epsilon^*.\pi\mu((\Rightarrow \epsilon^*)\,\S\,\zeta)\kappa)\sigma$$

Proof: Induction on the length of the expression sequences.

5.4.6 Statement sequences

$$(\mathcal{CS}^*[\![\mathtt{E}^*]\!]\tau\pi)\mu\zeta\kappa\sigma = \mathcal{E}^*[\![\mathtt{E}^*]\!](\tau,\mu)(\lambda\epsilon^*\sigma.\pi\mu\zeta\kappa\sigma)\sigma$$

Proof: Induction on the length of the expression sequences.

5.5 An operational semantics of the abstract machine

The abstract machine definition is denotational. However, except for the error rules, each instruction definition is of the form

$$\pi\mu\zeta\kappa\sigma = \pi'\mu'\zeta'\kappa'\sigma'$$

where $\pi, \mu, \zeta, \kappa, \sigma, \pi', u', \zeta', \kappa', \sigma'$ are given by the following grammar:

$$\begin{array}{rcl}
\pi & ::= & (\mathit{constant}\;\epsilon\,\pi) \mid (\mathit{fetch}\;\iota\,\pi) \mid (\mathit{assign}\;\iota\,\pi) \mid (\mathit{brf}\;\pi_0\,\pi_1) \mid (\mathit{brfng}\;\pi_0\,\pi_1) \\
 & & \mid (\mathit{goto}\;\pi) \mid (\mathit{close}\;\nu\,\pi_0\,\pi_1) \mid (\mathit{rts}) \mid (\mathit{catch}\;\pi_0\,\pi_1) \mid (\mathit{jmp}\;\nu) \\
 & & \mid (\mathit{jsr}\;\nu) \mid (\mathit{test\text{-}arg}\;\pi) \mid (\mathit{pop}\;\pi) \\
\mu & ::= & \langle\,\rangle \mid (\delta^* \cdot \mu') \\
\zeta & ::= & \langle\,\rangle \mid (\epsilon :: \zeta) \\
\epsilon & ::= & \mathit{inB}(\beta) \mid \mathit{inC}(c) \mid \mathit{inN}(\nu) \mid (\mathtt{closure}\;\nu\,\mu\,\pi_0) \\
\kappa & ::= & \mathit{init}_k \mid (\mathtt{cont}\;\pi\mu\zeta\kappa) \\
\sigma & ::= & \mathit{init}_s \mid \sigma[\delta^*/\epsilon^*]
\end{array}$$

Each step in the reduction sequence of $\pi\mu\zeta\kappa\sigma$ will have the form $\pi'u'\zeta'\kappa'\sigma'$ for some program π', display u', stack ζ', continuation κ', and variable store σ' as specified by this grammar. Therefore we can give the abstract machine an operational semantics by thinking of it as a 5-register machine and regarding each of the equations for the instruction combinators as rewriting rules[3]. The resulting abstract machine is shown in Figure 7. We can state the soundness of this machine as follows:

Theorem 1 *If* $(\pi, \mu, \zeta, \kappa, \sigma) \Longrightarrow (\pi', \mu', \zeta', \kappa', \sigma')$, *then* $\pi\mu\zeta\kappa\sigma \rightarrow \pi' u' \zeta' \kappa' \sigma'$.

where we use $\rightarrow$ to denote the four outermost β-reductions of the combinator π.

Proof (Informal): It can be shown that the machine is complete with respect to either call-by-value or call-by-name reduction [52], since these reduction strategies coincide on the terms manipulated by the machine, and the machine emulates these reduction strategies. □

5.6 Factorial example

In Figure 8 we recall the factorial example from Chapter 4. In Figure 9 we show the result of compiling the program. Comments (lines beginning with a ;) have been inserted to relate the compiled code and the original source program.

[3]While errors can be captured by the rewriting rules, for clarity of the presentation, we will no longer explicitly handle them. This is reasonable since the semantics of ν-LISP leaves the results on error unspecified.

Abstract Machine

Instructions

$$
\begin{array}{rcl}
((constant\ \epsilon\ \pi), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi, \mu, (\epsilon :: \zeta), \kappa, \sigma) \\
((fetch\ \iota\ \pi), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi, \mu, (\sigma(\mu \downarrow l) :: \zeta), \kappa, \sigma) \\
((assign\ \iota\ \pi), \mu, (\epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi, \mu, (\epsilon :: \zeta), \kappa, \sigma[\epsilon/(\mu \downarrow \iota)]) \\
((brf\ \pi_0\ \pi_1), \mu, (\epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & ((\epsilon = inB(false) \to \pi_1, \pi_0), \mu, \zeta, \kappa, \sigma) \\
((brfng\ \pi_0\ \pi_1), \mu, (\epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & ((\epsilon = inB(false) \to \pi_1, \pi_0), \mu, \zeta, \kappa, \sigma) \\
((goto\ \pi), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi, \mu, \zeta, \kappa, \sigma) \\
((close\ \nu\ \pi_0\ \pi_1), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi_1, \mu, (inF((\texttt{closure}\ \nu\ \mu\ \pi_0)) :: \zeta), \kappa, \sigma) \\
((rts), \mu, (\epsilon :: \zeta), (\texttt{cont}\ \pi'\mu'\zeta'\kappa'), \sigma) & \Longrightarrow & (\pi', u', (\epsilon :: \zeta'), \kappa', \sigma) \\
((catch\ \pi_0\ \pi_1), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi_1, \mu, \langle\,\rangle, (\texttt{cont}\ \pi_0\mu\zeta\kappa), \sigma) \\
((jmp\ \nu), \mu, (inF((\texttt{closure}\ \nu\ \mu'\ \pi')) :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi', ((alloc\ \nu\ \sigma) \cdot \mu'), \langle\,\rangle, \kappa, \sigma[\zeta/(alloc\ \nu\ \sigma)]) \\
((jsr\ \nu), \mu, (inF((\texttt{closure}\ \nu\ \mu'\ \pi')) :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi', ((alloc\ \nu\ \sigma) \cdot \mu'), \langle\,\rangle, \kappa, \sigma[\zeta/(alloc\ \nu\ \sigma)]) \\
((test\text{-}arg\ \pi), \mu, \zeta, \kappa, \sigma) & \Longrightarrow & (\pi, \mu, \zeta, \kappa, \sigma) \\
((pop\ \pi), \mu, (\epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi, \mu, \zeta, \kappa, \sigma)
\end{array}
$$

Primitives

$$
\begin{array}{rcl}
((sm\text{-}unprim_i\ \pi), \mu, (\epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi, \mu, ((unop_i\ \epsilon) :: \zeta), \kappa, \sigma) \\
((sm\text{-}binprim_i\ \pi), \mu, (\epsilon' :: \epsilon :: \zeta), \kappa, \sigma) & \Longrightarrow & (\pi, \mu, ((binop_i\ \epsilon\epsilon') :: \zeta), \kappa, \sigma)
\end{array}
$$

Figure 7: Actions of the Abstract Machine

```
((lambda (fact)
    (begin (set! fact
              (lambda (i a) (if (zero? i) a (fact (- i 1) (* a i)))))
           (fact 4 1)))
 0)
```

Figure 8: Iterative Factorial Program

```
;;; Push the argument to the outer closure
(constant 0
;;; Push outer closure
(close 1
  ;; Push factorial function
  (close 2
    ; Check value of i
    (fetch (0 . 2)
    (primitive zero? 1
    (brfng
      ; If i is zero return a
      (fetch (0 . 1)
      (rts))
    ; If i is not 0
    ; Push i - 1
    (fetch (0 . 2)
    (constant 1
    (primitive - 2
    ; Push i * a
    (fetch (0 . 1)
    (fetch (0 . 2)
    (primitive * 2
    ; Get factorial and jump to it.
    (fetch (1 . 1)
     (jmp 2)))))))))))
  ;; Assign factorial to local variable
  (assign (0 . 1)
  ;; Clear result of assign
  (pop
  ;; Push argument i to factorial
  (constant 4
  ;; Push argument a to factorial
  (constant 1
  ;; Get factorial and jump to it
  (fetch (0 . 1)
  (jsr 2)))))))
;;; Jump to outer closure
(jsr 1)))
```

Figure 9: Compiled Iterative Factorial Program

Chapter 6

Representation of Data

In the last chapter we presented an abstract machine for executing code produced by the ν-LISP compiler. Our overall goal is to map the abstract machine to a target machine that closely resembles contemporary machine architectures. In this chapter we will focus our attention on the representation of data from the abstract machine in the target machine.

The key insight in considering the representation of the abstract machine's data is that the ν-LISP language was designed to obey stack discipline. This means that the local stack, display, continuation, and variable store can all be threaded through a single, global stack.

To formalize the our data representations, we will introduce a hybrid of the abstract and the target machines which we will refer to as the stack machine. The stack machine will consist of the instruction set of the abstract machine and the pointers and global store of the target machine[1]. Using the formal description of the actions of the abstract machine instructions on the data components, we will define the stack machine instructions to manipulate its representations of the data components in an analogous manner. The formalisms of the definitions will then allow us to prove that, as long as data is represented properly, the stack machine will simulate the abstract machine.

[1]The full definition of the target machine (a.k.a. the register machine) will be given in chapter 7.

6.1 The stack machine

The stack machine machine consists of five components:

- π - the instruction to be executed.
- ϕ - a store that is manipulated like a stack. It contains the representations of environments, local stacks, and continuations from the abstract machine.
- *sp* - a pointer to the top of ϕ. This is where the abstract machine's local stack, ζ, is represented.
- *up* - a pointer indicating where the abstract machine's display, μ, and variable store, σ are represented in ϕ (we will refer to μ and σ together as the runtime environment). Our representation will merge the two components of the runtime environment for an efficient representation.
- *kp* - a pointer indicating where the current continuation, κ, is represented in ϕ.

Before we give the definition of the stack machine, in section 6.2 we will informally describe how each component of the abstract machine is represented in ϕ. Using these representations, we will show in section 6.3 how to derive the actions of the stack machine based upon the actions of the abstract machine. Finally, in sections 6.5 and 6.6 we will formalize the representations and prove that the actions of the stack machine simulate the actions of the abstract machine.

6.2 Informal discussion of data representations

6.2.1 Values

Values are represented in the stack machine the same way they are in the abstract machine with the exception of closures. A closure in the abstract machine is represented by:

$(\texttt{closure}\ \nu\, \mu\, \pi)$

where μ is the saved environment. The stack machine represents the same closure as:

(`closure` ν up π)

where up is a pointer into the stack which represents μ.

Additionally, we introduce a notation for extracting the various fields from closures:

- (`closure` ν up π)$.narg = \nu$
- (`closure` ν up π)$.env = up$
- (`closure` ν up π)$.body = \pi$

6.2.2 Local stacks

A local stack of size n from the abstract machine is represented by n contiguous locations in the stack machine's stack. It is specified by giving upper and lower bounds. The representation of

$$\zeta = \langle \epsilon_n, \ldots, \epsilon_1 \rangle$$

is:

p :	ϵ_n	representation of the top value in ζ
	$\vdots$	
	ϵ_1	representation of the bottom value in ζ
p' :		

This diagram is intended to mean that p and p' are pointers into the stack, that p points higher in the stack than p', and that the values in the stack from position p, moving downward, are ϵ_n, ϵ_{n-1}, $\ldots$, ϵ_1.

6.2.3 Runtime environments

In the abstract machine, runtime environments are split between runtime displays and the variable store. To add n values to a runtime environment, n new cells are allocated from the store, the cells are updated with the values, and the display is updated with the pointers to the cells.

The obvious stack representation of this situation is to represent displays as sequences of pointers into the store. This approach requires an unnecessary level of indirection because the values can actually just be written to and read directly from the stack. The trick is to represent runtime environment extensions entirely on the stack. When invoking a closure, $(\mathtt{closure}\ \nu\, \mu\, \pi)$, on arguments $\epsilon_1, \ldots, \epsilon_\nu$, the representation of the resulting environment

$$\mu' = (\langle \delta_\nu, \ldots, \delta_1 \rangle \cdot \mu) \qquad \sigma' = \sigma[\langle \epsilon_\nu, \ldots, \epsilon_1 \rangle / \langle \delta_\nu, \ldots, \delta_1 \rangle]$$

is:

p :	$(\mathtt{closure}\ \nu\ up\ \pi)$	static chain pointer
	ϵ_ν	representation of δ_ν and its cell containing ϵ_ν
	$\vdots$	
	ϵ_1	representation of δ_1 and its cell containing ϵ_1

The stack positions $p - 1$ through $p - \nu - 1$ now play a dual role of representing the store pointers in μ and the corresponding cells containing the arguments in σ. The closure on the top of the stack is used as a static chain pointer (via up)[2] and as an argument count. For the representation to be correct, the static chain pointer must represent the rest of closure's environment contours in the same manner.

6.2.4 Continuations

In the abstract machine, continuations are represented by

$$\kappa = (\mathtt{cont}\ \pi \mu \zeta \kappa')$$

where μ, ζ and κ' are the saved machine state and π is the code to be executed when the procedure returns (i.e. the return address). In our representation of continuations, the continuation code is wrapped in a closure which gives consistent access to code stored on the stack. Its only use is to access the continuation code when the continuation is invoked. We represent the continuation by a pointer p:

[2]We use closures as static chain pointers because they are already on top of the stack when performing environment extensions. Alternatively, we could pop the closures and push their stored environment pointers.

p :	up	representation of saved μ
	kp	representation of saved κ'
	(`closure` 1 up π)	saved return address
	ϵ_n	top of saved ζ
	$\vdots$	
	ϵ_1	bottom of saved ζ
(max up kp) :		

6.2.5 Machine states

We now combine the above representation strategies to show how a typical state of the abstract machine is represented.

Under normal operation, the local stack is on represented on the top of the stack machine's stack followed by the representation of the current runtime environment and finally the representation of the current continuation. Hence

$$\begin{aligned}
\mu &= (\langle \delta_\nu, \ldots, \delta_1 \rangle \cdot \mu') \\
\zeta &= \langle \epsilon_n, \ldots, \epsilon_1 \rangle \\
\kappa &= (\texttt{cont}\, \pi \mu' \zeta' \kappa') \\
\sigma &= \sigma'[\langle \epsilon'_\nu, \ldots, \epsilon'_1 \rangle / \langle \delta_\nu, \ldots, \delta_1 \rangle]
\end{aligned}$$

is represented by the pointers up, sp, and kp:

sp :	ϵ_n	representation of ζ
	$\vdots$	
	ϵ_1	
up :	(`closure` ν up' π')	representation of μ and σ
	ϵ'_ν	
	$\vdots$	
	ϵ'_1	
kp :	up''	representation of κ
	kp'	
	(`closure` 1 up'' π'')	
	$\vdots$	

When setting up for a call to a procedure in a non-tail position, a *catch* is executed which creates a continuation containing the current display (to be restored when the procedure returns). Before the procedure is invoked, the state of the abstract machine:

$$
\begin{aligned}
\mu &= (\langle \delta_\nu, \ldots, \delta_1 \rangle \cdot \mu') \\
\zeta &= \langle \epsilon_n, \ldots, \epsilon_1 \rangle \\
\kappa &= (\texttt{cont}\, \pi \mu \zeta' \kappa') \\
\sigma &= \sigma'[\langle \epsilon'_\nu, \ldots, \epsilon'_1 \rangle / \langle \delta_\nu, \ldots, \delta_1 \rangle]
\end{aligned}
$$

is represented as:

sp :	ϵ_n	representation of ζ
	$\vdots$	
	ϵ_1	
kp :	up	
	kp'	
	(`closure` $1\ up\ \pi$)	representation of κ
	$\vdots$	
up :	(`closure` $\nu\ up'\ \pi'$)	representation of μ and σ
	ϵ'_ν	
	$\vdots$	
	ϵ'_1	

In this representation, the continuation frame is above the runtime environment frame. The values above the continuation will constitute the new environment frame once the call is made and the representation of the machine state will be as originally shown.

6.3 Deriving the actions of the stack machine

With this understanding of the of the representation of data, we can now derive the actions of each instruction for the stack machine. The goal is to give a definition for each instruction in such a way that the correspondence between the abstract and stack machine is preserved through the execution of the instruction.

For example, consider the abstract machine's actions on a *jsr* as shown at the top of Figure 10. A *jsr* is used to call procedures not in tail position. Prior to evaluating the arguments, a continuation is pushed by a *catch* instruction (specified in section 5.2.3). The values on the local stack are the arguments to the procedure, except the top value which is the procedure itself. The representation of the stack state is shown in the left half of Figure 10.

Abstract machine actions on a *jsr*

$$\begin{array}{l}((\mathit{jsr}\ \nu), \mu, ((\texttt{closure}\ \nu\, \mu'\, \pi') :: \zeta), \kappa, \sigma) \Longrightarrow \\ \quad (\pi', ((\mathit{alloc}\ \nu\, \sigma) \cdot \mu'), \langle\, \rangle, \kappa, \sigma[\zeta/(\mathit{alloc}\ \nu\, \sigma)])\end{array}$$

	Stack machine state before a *jsr*		Stack machine state after a *jsr*
sp :	($\texttt{closure}\ \nu_2\ \mathit{up}_2\ \pi_2$)	up', sp' :	($\texttt{closure}\ \nu_2\ \mathit{up}_2\ \pi_2$)
	ϵ_{ν_2}		ϵ_{ν_2}
	$\vdots$		$\vdots$
	ϵ_1		ϵ_1
kp :	up	kp' :	up
	kp_1		kp_1
	($\texttt{closure}\ 1\ \mathit{up}\ \pi_1$)		($\texttt{closure}\ 1\ \mathit{up}\ \pi_1$)
	$\vdots$		$\vdots$
up :	($\texttt{closure}\ \nu_0\ \mathit{up}_0\ \pi_0$)	up :	($\texttt{closure}\ \nu_0\ \mathit{up}_0\ \pi_0$)

Figure 10: Executing a *jsr* Instruction

When the call does occur in the abstract machine, ν locations are allocated from the store, the arguments are bound to those locations, the runtime display is updated with those locations, and the procedure's code executes with an empty local stack.

The simulation of these actions by the stack machine is shown in the right half of Figure 10. The values up', sp', and kp' represent the values of the machine registers in the new state. The only register whose value changes is the environment pointer, up. Updating up has the effect of implicitly allocating ν mutable cells, binding the arguments to the cells, and updating the runtime display with these cells. The closure value at the top of the stack is now used only as a static chain pointer.

Continued application of this reasoning yields the definition of the stack machine shown in Figure 11.

Stack Machine:

Instructions

$$
\begin{array}{rcl}
((\mathit{goto}\ \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp, kp, \phi) \\
((\mathit{constant}\ \epsilon\, \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp{+}1, kp, \phi[\epsilon/\ sp{+}1]) \\
((\mathit{fetch}\ \iota\, \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp{+}1), kp, \phi[(\texttt{find}\ \iota\ up\ \phi)/\ sp{+}1]) \\
((\mathit{assign}\ \iota\, \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp), kp, \phi[(\phi\ sp)/(\texttt{find}\ \iota\ up\ \phi)]) \\
((\mathit{brf} \pi_0 \pi_1), up, sp, kp, \phi) & \Longrightarrow & (((\phi\ sp) = inB(false) \rightarrow \pi_1, \pi_0), up, sp{-}1, kp, \phi) \\
((\mathit{close}\ \nu\, \pi_0\, \pi_1), up, sp, kp, \phi) & \Longrightarrow & (\pi_1, up, sp{+}1, kp, \phi[(\texttt{closure}\ \nu\ up\ \pi_0)/\ sp{+}1)) \\
((\mathit{rts}), up, sp, kp, \phi) & \Longrightarrow & ((\phi\,(kp{-}2)).code, (\phi\ kp), kp{-}2, (\phi\,(kp{-}1)), \phi[(\phi\ sp)/\ kp{-}2]) \\
((\mathit{catch}\ \pi_0\, \pi_1), up, sp, kp, \phi) & \Longrightarrow & (\pi_1, up, sp{+}3, sp{+}3, \phi[kp\,/\ sp{+}1, up\,/\ sp{+}2, \pi_0/\ sp{+}3]) \\
((\mathit{jmp}\ \nu), up, sp, kp, \phi) & \Longrightarrow & ((\phi\ sp).body, (kp{+}\nu+1), (kp{+}\nu+1), kp, \\
 & & \quad \phi(\texttt{slide}\,(\nu+1)\,(sp{-}\nu)\,(kp{+}1))) \\
((\mathit{jsr}\ \nu), up, sp, kp, \phi) & \Longrightarrow & ((\phi\ sp).body, sp, sp, kp, \phi) \\
((\mathit{pop}\ \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp{-}1, kp, \phi)
\end{array}
$$

Primitives

$$
\begin{array}{rcl}
((\mathit{sm\text{-}unprim}_i\ \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp, kp, \phi[(\mathit{unop}_i\,(\phi\ sp))/\ sp]) \\
((\mathit{sm\text{-}binprim}_i\ \pi), up, sp, kp, \phi) & \Longrightarrow & (\pi, up, sp, kp, \phi[(\mathit{binop}_i\,(\phi\ sp{-}1)\,(\phi\ sp))/\ sp{-}1])
\end{array}
$$

Auxiliaries

$$
\begin{array}{rcl}
(\texttt{find}\ (n, m)\ up\ \phi) & = & (n = 0) \rightarrow (up - m - 1), \\
 & & \quad (\texttt{find}\ (n-1, m)\,((\phi\ up).env)\,\phi) \\
(\texttt{slide}\ n\, f\, t) & = & (n = 1) \rightarrow [(\phi\, f)/t], \\
 & & \quad [(\phi\, f)/t](\texttt{slide}\ n\,(f+1)\,(t+1))
\end{array}
$$

Figure 11: Actions of the Stack Machine

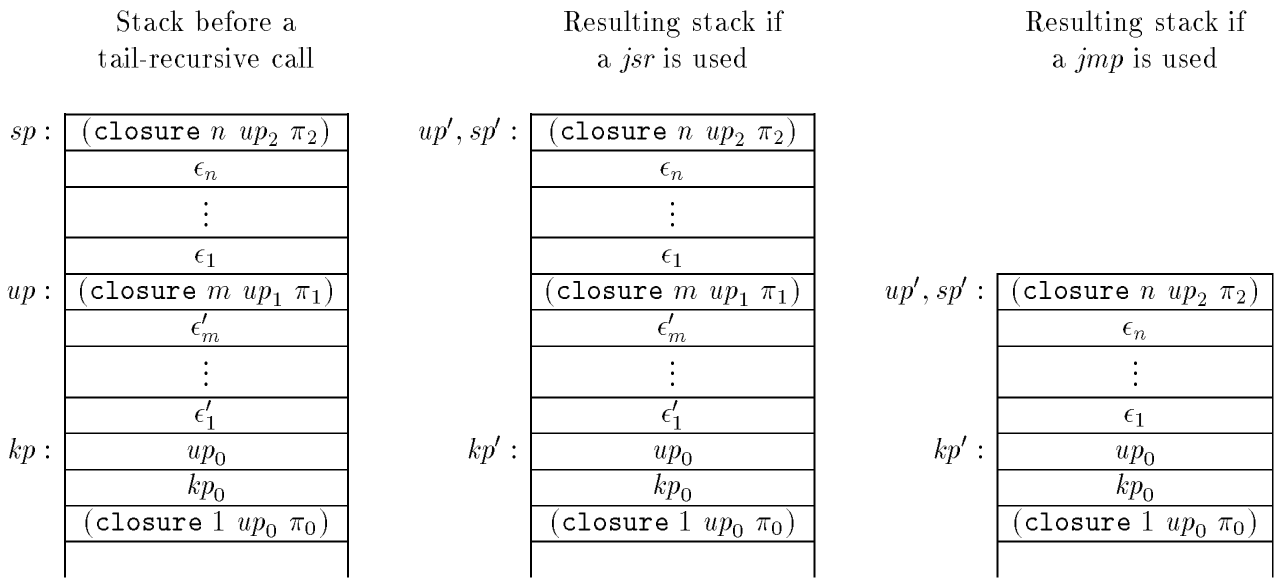

Figure 12: Making Tail-recursive Calls

6.4 Tail recursion

When calling a procedure that is not in a tail position, we create a new continuation, collect the arguments, and then execute a *jsr*. For a procedure in tail position, we do not create a new continuation, we just collect the arguments and then jump to the procedure (shown the first column of Figure 12). Jumping to the procedure can occur using either a *jsr* or *jmp* instruction. If a *jsr* is used, the new environment frame sits on top of the current one and the procedure executes as before (shown in the second column of Figure 12).

However, if a *jmp* is used this means the compiler determined that the topmost environment frame can no longer be accessed. By reclaiming this space, we can avoid unnecessarily overflowing the stack during nested tail recursive calls. Therefore, we slide the new environment frame over the current environment frame before executing the procedure (shown in the third column of Figure 12).

6.5 Formalizing the data representations

We now formalize the representation of the abstract machine components described in section 6.2. We do this using *storage layout relations* as described in [78]. In general a storage layout relation is a ternary relation:

$$\Gamma \models sc \simeq ac$$

which reads "under store assumptions Γ, stack components sc represent abstract components ac". We will define several different storage layout relations to describe the representations of the abstract machine components. To differentiate between the variations, we will add a subscript to $\simeq$.

6.5.1 Store assumptions

Abstract machine store pointers are implicit in the stack machine. To relate the two machines, we need to track which locations of the stack machine's stack represent store locations and which store locations they represent. To do this, we will associate with each stack location a store assumption indicating either that the stack location does not represent a store location or that it does represent a store location and which location it represents. In particular, for stack location p, store assumption a can be one of two alternatives:

$$a \quad ::= \quad \bot \mid \delta$$

- The constant $\bot$ meaning that p does not implicitly represent a store location in the abstract machine.
- The value δ meaning that the stack location p represents store location δ in the abstract machine.

All data for the stack machine lives on the stack from 0 to sp. The relationship between machine states includes a sequence of $sp+1$ store assumptions Γ:

	Store assumptions Γ	Stack machine stack ϕ
sp :	a_{sp}	v_{sp}
$sp-1$:	a_{sp-1}	v_{sp-1}
$\vdots$	$\vdots$	$\vdots$
1 :	a_1	v_1
0 :	a_0	v_0

To find if stack location p in the stack machine represents a store location in the abstract machine, we look it up in Γ (i.e. we take $\Gamma \uparrow p$). In the following example:

	Store assumptions Γ	Stack machine stack ϕ		Abstract machine store σ
3 :	$\perp$	v_3	3 :	v_0
2 :	$\delta' = 1$	v_2	2 :	unknown
1 :	$\perp$	v_1	1 :	v_2
0 :	$\delta = 3$	v_0	0 :	unknown

stack location 0 represents store location 3 ($\Gamma \uparrow 0 = 3$). For the stack machine to simulate the abstract machine under Γ, the value of ϕ at location 0 must agree with σ at location 3 (similarly for stack location 2 and store location 1).

6.5.2 Storage layout relations

Values

A stack machine value, ϵ_s, represents an abstract machine value, ϵ_a if

- The two values are the same.
- The two values are closures with the same number of arguments, the same code, and corresponding environments.

Formally:

$$\frac{\epsilon_s = \epsilon_a}{\Gamma \models (\epsilon_s, up, \phi) \simeq_V (\epsilon_a, \mu, \sigma)}$$

$$\frac{up' \leq up \quad \mu' \subseteq \mu \qquad \Gamma \models (up', \phi) \simeq_U (\mu', \sigma)}{\Gamma \models ((\texttt{closure}\ \nu\ up'\ \pi), up, \phi) \simeq_V ((\texttt{closure}\ \nu\, \mu'\, \pi), \mu, \sigma)}$$

Local stacks

We represent a local stack

$\zeta = \langle \epsilon_n, \ldots, \epsilon_1 \rangle$

by n contiguous locations on the stack, $p' \ldots p$, containing representations of the values. Additionally, each position in the representation ζ does not implicitly represent a store pointer so the store assumption about each location should be $\perp$.

	Store assumptions Γ	Stack machine stack ϕ	
$p:$	$\perp$	ϵ'_n	value corresponding to ϵ_n
$\vdots$	$\vdots$	$\vdots$	
	$\perp$	ϵ'_0	value corresponding to ϵ_1
$p':$			

We formalize this picture as:

$$\frac{p \geq p' \quad \#\epsilon^* = p - p' \quad \forall\ (0 \leq i < p - p')\ \Gamma(p - i) = \perp \qquad \forall\ (0 \leq i < p' - p)\ \Gamma \models ((\phi\, p - i), up, \phi) \simeq_V ((\epsilon^* \downarrow i), \mu, \sigma)}{\Gamma \models (p, p', up, \phi) \simeq_Z (\epsilon^*, \mu, \sigma)}$$

Runtime environments

The initial environment is represented by the initial environment pointer, $init_{up}$:

$$\frac{up = init_{up} \quad \mu = \langle\,\rangle}{\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)}$$

A runtime environment extension

$$\mu = (\langle \delta_\nu, \ldots, \delta_1 \rangle \cdot \mu') \qquad \sigma = \sigma'[\langle \epsilon_\nu, \ldots, \epsilon_1 \rangle / \langle \delta_\nu, \ldots, \delta_1 \rangle]$$

is represented by the $\nu + 1$ positions down from the environment pointer up. Position up contains a static chain pointer which represents u' and the positions $up-1 \ldots up-\nu$ contain values corresponding to values $\epsilon_\nu \ldots \epsilon_1$. The positions $up-1 \ldots up-\nu$ also implicitly represent store pointers $\delta_\nu \ldots \delta_1$ which is reflected by the store assumptions:

	Store assumptions Γ	Stack machine stack ϕ	
up :	$\perp$	(`closure` ν up' π)	static chain pointer
$up-1$:	δ_ν	ϵ_ν	value corresponding to ϵ_ν
$\vdots$	$\vdots$	$\vdots$	
$up-\nu-1$:	δ_1	ϵ_1	value corresponding to ϵ_1

Additionally, for stack discipline, the static chain pointer must point below the current environment frame. These conditions are formalized as:

$$\frac{\begin{array}{c}(\phi\ up).env < up-n-1 \quad \mu = (\langle \delta_n, \ldots, \delta_1 \rangle \cdot \mu') \\ (\Gamma \uparrow up = \perp) \quad \forall\ (1 \leq i \leq n)\ (\Gamma \uparrow (up-i)) = \delta_i \\ \forall\ (1 \leq i \leq n)\ \Gamma \models ((\phi\ up-i), up, \phi) \simeq_V ((\sigma\, \delta_i), \mu, \sigma) \\ \Gamma \models ((\phi\ up).env, \phi) \simeq_U (\mu', \sigma)\end{array}}{\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)}$$

Continuations

The initial continuation is represented by the initial continuation pointer, $init_{kp}$:

$$\frac{kp = init_{kp} \quad \kappa = init_k}{\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)}$$

A continuation

$$\kappa = (\texttt{cont}\, \pi\mu\zeta\kappa')$$

is represented by the $\#\zeta + 3$ positions down from location kp. Position kp contains the saved environment pointer that represents μ. Position $kp-1$ contains the saved continuation pointer that represents κ'. Position $kp-3$ contains the saved return address which corresponds to π. Finally, the positions $kp-3 \ldots kp-\#\zeta-4$ represent the saved local stack ζ. None of these positions implicitly represent a store pointer so their corresponding store assumptions are $\perp$.

	Store assumptions Γ	Stack machine stack ϕ	
$kp:$	$\perp$	up	saved representation of μ
$kp-1:$	$\perp$	kp	saved representation of κ'
$kp-2:$	$\perp$	$(\texttt{closure}\ 1\ up\ \pi)$	saved return address
$kp-3:$	$\perp$	ϵ_n	top of saved ζ
$\vdots$	$\vdots$	$\vdots$	
$kp-\#\zeta-3:$	$\perp$	ϵ_1	bottom of saved ζ

Again, we add constraints that require all pointers to point down and formalize the conditions for continuations as:

$$\frac{\begin{array}{c} \kappa = (\texttt{cont}\ \pi\mu\zeta\kappa') \\ (\Gamma \uparrow kp) = (\Gamma \uparrow (kp-1)) = (\Gamma \uparrow (kp-2)) = \perp \\ (\phi\ kp) < kp \quad (\phi\,(kp-1)) < kp \quad (\phi\,(kp-2)).body = \pi \\ \Gamma \models ((\phi\ kp), \phi) \simeq_U (\mu, \sigma) \\ \Gamma \models ((\phi\,(kp-1)), \phi) \simeq_K (\kappa', \sigma) \\ \Gamma \models (kp-3, kp-\#\zeta-4, (\phi\ kp), \phi) \simeq_Z (\zeta, \mu, \sigma) \end{array}}{\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)}$$

Machine states

We can now define the correspondence between abstract and stack machine states. In section 6.2 we described how an abstract machine state, μ, ζ, κ, and σ, may be represented in two ways.

The first representation is used when the current display is not contained within the current continuation. In this case, the representation of ζ is on top of the stack, followed by the representation of the current environment, followed by the representation of κ.

The second representation is used after a *catch* has been executed and the current continuation contains the current display. In this case the machine state is represented with ζ on top of the stack, followed by the representation of κ, followed by the representation of the current environment.

Most of the instructions will work with either of the representation, as long as each component of the abstract machine is represented properly with ζ on top of the stack. For the *jmp* and *jsr* it does matter. The *jmp* instruction expects to have the environment on top of the continuation while the *jsr* expects the continuation on top of the environment.

Additionally, we restrict the representation of environments such that each location in the variable store is represented by a single location on the stack. We formalize this by defining a class of store assumptions, Γ, that contain only one occurrence of any particular store pointer δ.

Definition 2 *We say that a sequence of store assumptions* Γ *is an* injection *if* $\forall\ (0 \leq i < \#\Gamma)\ \forall\ (0 \leq j < \#\Gamma)$ *if* $(\Gamma \uparrow i) = \delta$ *and* $(\Gamma \uparrow j) = \delta$ *then* $i = j$

Combining all of these requirements gives the following formalization of abstract and stack machine correspondence:

$$\frac{\Gamma \text{ is an injection} \quad sp \geq up \quad up \geq kp \qquad \Gamma \models (up, \phi) \simeq_U (\mu, \sigma) \qquad \Gamma \models (sp, up, up, \phi) \simeq_Z (\zeta, \mu, \sigma) \qquad \Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)}{\Gamma \models ((jmp\ \nu), up, sp, kp, \phi) \simeq ((jmp\ \nu), \mu, \zeta, \kappa, \sigma)}$$

$$\frac{\Gamma \text{ is an injection} \quad sp \geq kp \quad kp \geq up \qquad \Gamma \models (up, \phi) \simeq_U (\mu, \sigma) \qquad \Gamma \models (sp, kp, up, \phi) \simeq_Z (\zeta, \mu, \sigma) \qquad \Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)}{\Gamma \models ((jsr\ \nu), up, sp, kp, \phi) \simeq ((jsr\ \nu), \mu, \zeta, \kappa, \sigma)}$$

$$\frac{\Gamma \text{ is an injection} \quad sp \geq kp \quad sp \geq up \qquad \Gamma \models (up, \phi) \simeq_U (\mu, \sigma) \qquad \Gamma \models (sp, (max\ up\ kp), up, \phi) \simeq_Z (\zeta, \mu, \sigma) \qquad \Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)}{\Gamma \models (\pi, up, sp, kp, \phi) \simeq (\pi, \mu, \zeta, \kappa, \sigma)}$$

6.6 Verification of the data representations

In this section we will show that stack machine simulates the actions of the abstract machine. To do this, we will first show how to extract from a abstract machine state the store assumptions required for its representation. We can then show that as long as the stack machine and the abstract machine are in corresponding states under the extracted store assumptions, they will remain in corresponding states throughout their execution.

6.6.1 Extracting store assumptions

To verify that our representations of the various components of the abstract are correct, we need to know which locations on the stack implicitly represent store pointers and which store pointers they represent. In this section we define how to extract that information from the various components of the abstract machine for their representations. This approach will determine from the abstract machine state how it must be represented on the stack and will sequentially construct the necessary register assumptions about the stack from the top down. This construction is somewhat complicated by the existence of garbage environment frames on the runtime stack. To handle these frames properly, we must parameterize our construction by the runtime stack and a pointer indicating where in the runtime stack we are collecting store assumptions and then explicitly check for garbage.

Machine states

Our goal is to construct from the abstract machine components the store assumptions that must go along with their representations. As we've seen, an abstract machine state is represented in either of two ways. In both representations, the local stack is represented on the top of the global stack. This means that the store assumptions for the abstract machine state must be the store assumptions for the local stack followed by the store assumptions for the rest of the stack. We now define a function that will collect the store assumptions from an abstract machine state:

$$(collect(\zeta, \mu, \kappa)\,(p, \phi)) = (collect_z\ \zeta)\,\S(collect_{u,k}\,(\mu, \kappa)\,(p - \#\zeta, \phi))$$

The rest of the stack is made up of environment frames and continuation frames that represent the runtime environments and continuations. We will extract the register assumptions about each frame until we have collected the assumptions about the entire stack (i.e. when we have reached the initial display and continuation).

The display's topmost contour is the topmost environment frame that we will find on the stack and the continuation representation is the first continuation frame we will find on the stack. To determine which of the two frames will appear first, we examine the continuation and display.

If the current continuation contains the current display, the topmost continuation frame must appear above the environment frame. Thus we collect the assumptions about the continuation frame and iterate the process with the saved continuation (appearing within the current continuation).

If the current display is not contained within the current continuation, the environment frame must appear above the continuation frame. Thus we collect the assumptions that must go along with the topmost environment frame. We note that the next environment frame that we will encounter on the stack is the one saved within the current environment. Hence to extract the register assumptions about the rest of the stack, we extract the assumptions about the current continuation and the display contained within the current environment.

When we get the to initial continuation and the initial display, nothing else can appear on the stack so there are no more assumptions.

To handle garbage, we note that garbage frame can only occur under the current environment frame. Once we collect the assumptions about the environment frame, we collect assumptions about any garbage that may follow it on the runtime stack, and then collect the rest of the assumptions about the stack.

$$(\mathit{collect}_{u,k}\,(\mu,\kappa)) = \begin{cases} \text{if } \kappa = (\texttt{cont}\ \pi\mu'\zeta\kappa') \text{ and } u = u' \text{ then} \\ \quad (\mathit{collect}_k\ \kappa)\,\S(\mathit{collect}_{u,k}\,(\mu,\kappa')\,(p-(\#\zeta+3),\phi)) \\ \text{if } \kappa = (\texttt{cont}\ \pi\mu'\zeta\kappa') \text{ and } u = (S\cdot\mu') \text{ and } \phi(p) = (\texttt{closure}\ \nu\ \mathit{up}\ \pi) \\ \quad\ \text{and } (\mathit{collect}_g\,(p-(\nu+1),\mathit{up},\phi) = S \text{ then} \\ \quad\quad (\mathit{collect}_u\ \mu)\,\S\,S\,\S(\mathit{collect}_{u,k}\,(\mu',\kappa)\,(p-(\nu+1+\#S),\phi)) \\ \text{if } \mu = \langle\,\rangle \text{ and } \kappa = \mathit{init}_k \text{ then } \langle\,\rangle \\ \text{otherwise } \mathit{error} \end{cases}$$

Local stacks

Given an abstract stack ζ, we know that the store assumptions for its representation should all be $\perp$:

$$(\mathit{collect}_z\ \zeta) = \langle \perp_1, \ldots, \perp_{\#\zeta} \rangle$$

Environment frames

To extract the store assumptions about the top contour of a display $\mu = (\langle \delta_n, \ldots, \delta_1 \rangle \cdot \mu')$, we know that the store assumptions for its representation by the stack machine should be $\perp$ for the static chain pointer followed by $\delta_n, \ldots, \delta_1$:

$$(collect_u\ \mu) = \begin{cases} \text{if } \mu = \langle\,\rangle \text{ then } \langle\,\rangle \\ \text{if } \mu = (\langle \delta_n, \ldots, \delta_1 \rangle \cdot \mu') \text{ then } \perp ::\langle \delta_n, \ldots, \delta_1 \rangle \\ \text{otherwise } \mathit{error} \end{cases}$$

Continuations

For a continuation $\kappa = (\texttt{cont}\ \pi\mu\zeta\kappa')$, the store assumptions for the topmost frame of its representation should be $\langle \perp, \perp, \perp \rangle$ for the saved environment pointer, saved continuation pointer, and saved return address. Additional, we need to collect the store assumptions about the saved local stack:

$$(collect_k\ \kappa) = \begin{cases} \text{if } \kappa = \mathit{init}_k \text{ then } \langle\,\rangle \\ \text{if } \kappa = (\texttt{cont}\ \pi\mu\zeta\kappa') \text{ then } \langle \perp, \perp, \perp \rangle\ \S(collect_z\ \zeta) \\ \text{otherwise } \mathit{error} \end{cases}$$

Garbage environment frames

Detecting garbage is done through the use of three parameters: p - the current position in the stack we are gathering assumptions about, p' - the position of the next legal environment frame in the stack, and ϕ - the stack we are gathering assumptions about. If $p = p'$ then there are no garbage environment frames to gather so we are done. If p points to an environment frame and $p \neq p'$ then the frame must be garbage and we give $\perp$ as the assumption for each position in the frame (since the frame is garbage, the memory has been implicitly deallocated). Finally if p does not point to an environment frame, we gather no assumptions.

$$(collect_g\ (p, p', \phi)) = \begin{cases} \text{if } p = p' \text{ then } \langle\,\rangle \\ \text{if } \phi(p) = (\texttt{closure}\ \nu\ \mathit{up}\ \pi) \text{ then } (\perp_0, \ldots, \perp_\nu)\ \S(collect_g(p - (\nu + 1), p', \phi) \\ \text{otherwise } \langle\,\rangle \end{cases}$$

6.6.2 Proof of simulation

We now verify that the stack machine simulates the abstract machine. We begin by stating some useful lemmas about the storage layout relations and machine correspondence. In these lemmas we will use the notation $\phi =_p \phi'$ to denote the proposition $\forall x, 0 \leq x \leq p \Longrightarrow \phi(x) = \phi'(x)$. Similarly we will use the notation $\Gamma =_p \Gamma'$ to denote the proposition $\forall x, 0 \leq x \leq p \Longrightarrow (\Gamma \uparrow x) = (\Gamma' \uparrow x)$.

The most important of these lemmas states that if we change the store assumptions about locations in the stack above a pointer, it does not change the correspondence below that pointer (since all pointers point down).

Lemma 3 *For store assumptions* Γ

1. *If* $\Gamma =_p \Gamma'$ *and* $\Gamma \models ((\phi\, p), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$ *then* $\Gamma' \models ((\phi\, p), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$
2. *If* $\Gamma =_{up} \Gamma'$ *and* $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$ *then* $\Gamma' \models (up, \phi) \simeq_U (\mu, \sigma)$.
3. *If* $\Gamma =_{sp} \Gamma'$ *and* $\Gamma \models (sp, (max\ up\ kp), up, \phi) \simeq_Z (\zeta, \mu, \sigma)$ *then* $\Gamma' \models (sp, (max\ up\ kp), up, \phi) \simeq_Z (\zeta, \mu, \sigma)$.
4. *If* $\Gamma =_{kp} \Gamma'$ *and* $\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)$ *then* $\Gamma' \models (kp, \phi) \simeq_K (\kappa, \sigma)$.

Proof: By induction on the pointers into ϕ. $\square$

The next lemma similarly states that changing locations in the stack above a pointer does not change the quantity that the pointer represents.

Lemma 4 *For stack machine stack* ϕ

1. *If* $\phi =_p \phi'$ *and* $\Gamma \models ((\phi\, p), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$ *then* $\Gamma \models ((\phi\, p), up, \phi') \simeq_V (\epsilon, \mu, \sigma)$
2. *If* $\phi =_{up} \phi'$ *and* $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$ *then* $\Gamma \models (up, \phi') \simeq_U (\mu, \sigma)$.
3. *If* $\phi =_{sp} \phi'$ *and* $\Gamma \models (sp, (max\ up\ kp), up, \phi) \simeq_Z (\zeta, \mu, \sigma)$ *then* $\Gamma \models (sp, (max\ up\ kp), up, \phi') \simeq_Z (\zeta, \mu, \sigma)$.

4. *If* $\phi =_{kp} \phi'$ *and* $\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)$ *then* $\Gamma \models (kp, \phi') \simeq_K (\kappa, \sigma)$.

Proof: By induction on the pointers into ϕ. $\Box$

We also show the correctness of the auxiliary `find` given the proper representation of a display. In particular, we verify that given a lexical address, following the static chain pointer leads to a position in the stack which implicitly represents the store cell saved in the display at the same lexical address.

Lemma 5 *If* $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$ *then* $(\Gamma \uparrow (\texttt{find}\ \iota\ up\ \phi)) = (\mu \downarrow \iota)$

Proof: by induction on ι.

We now address the issue of performing an assignment. This is a little trickier than previous proofs because we will be overwriting the stack machine's stack. We are helped by the stack discipline restrictions placed upon the values that can be assigned to a lexical address. In particular, when assigning a value to a lexical address the value is either immediate data or a closure whose saved environment is contained within the environment being assigned to. We formalize this notion by this definition:

Definition 3 *An abstract machine value* ϵ *is* assignable *to lexical address* ι *in display* μ *if* $(\mu \downarrow \iota)$ *is not an error and either:*

- ϵ *is not a closure*
- $\epsilon = (\texttt{closure}\ \nu\, \mu'\, \pi)$ *and* $\mu' \subseteq_C \mu/\iota$

Finally, the next lemma shows shows that if the machines are in corresponding states then updating the runtime environments with corresponding values leaves the machines in corresponding states.

Lemma 6 *If* Γ *is an injection,* $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$, $\Gamma \models (\epsilon_s, up, \phi) \simeq_V (\epsilon_a, \mu, \sigma)$, *and* ϵ_a *is assignable to* ι *in* μ *then for* $\phi' = \phi[\epsilon_s/(\texttt{find}\ \iota\ up\ \phi)]$ *and* $\sigma' = \sigma[\epsilon_a/(\mu \downarrow \iota)]$*:*

1. *If* $\Gamma \models (\epsilon'_s, up', \phi) \simeq_V (\epsilon'_a, \mu', \sigma)$ *then* $\Gamma \models (\epsilon'_s, up', \phi') \simeq_V (\epsilon'_a, \mu', \sigma')$

2. *If* $\Gamma \models (up', \phi) \simeq_U (\mu', \sigma)$ *then* $\Gamma \models (up', \phi') \simeq_U (\mu', \sigma')$

3. *If* $\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)$ *then* $\Gamma \models (kp, \phi') \simeq_U (\kappa, \sigma')$

4. *If* $\Gamma \models (p, p', up', \phi) \simeq_Z (\zeta, \mu', \sigma)$ *then* $\Gamma \models (p, p', up', \phi') \simeq_Z (\zeta, \mu', \sigma')$

Proof: by induction on ι.

Definition 4 *We say a stack machine state* (π, up, sp, kp, ϕ) corresponds *to an abstract machine state* $(\pi', \mu, \zeta, \kappa, \sigma)$ *(written* $(\pi, up, sp, kp, \phi) \simeq (\pi', \mu, \zeta, \kappa, \sigma)$*) if and only if collect(* $\zeta, \mu, \kappa)\,(sp, \phi) \models (\pi, up, sp, kp, \phi) \simeq (\pi', \mu, \zeta, \kappa, \sigma)$

Now we can show the main theorem, which asserts that the stack machine correctly simulates the abstract machine:

Theorem 2 *Let* A_1 *and* A_2 *be states of the abstract machine and* S_1 *and* S_2 *be states of the stack machine. If* $S_1 \simeq A_1$*,* $A_1 \Longrightarrow A_2$*, and* $S_1 \Longrightarrow S_2$*, then* $S_2 \simeq A_2$*.*

Proof: By analysis of each instruction. For brevity, we only show the proof for two instructions. The other instructions can be proved in a similar manner.

- Assume we have $\Gamma \models ((constant\ \epsilon\ \pi), up, sp, kp, \phi) \simeq ((constant\ \epsilon\ \pi), \mu, \zeta, \kappa, \sigma)$. We must show that

  $$\perp :: \Gamma \models (\pi, up, sp+1, kp, \phi') \simeq (\pi, u, (\epsilon :: \zeta), \kappa, \sigma)$$

  where $\phi' = \phi[\epsilon/\ sp+1]$.

  We consider each of the conditions for $\simeq$ in turn:

  1. By assumption $\pi = \pi$.
  2. By assumption Γ is an injection hence $\perp :: \Gamma$ is an injection.

3. By assumption $sp \geq kp$ hence $sp+1 \geq kp$.

4. By assumption $sp \geq up$ hence $sp+1 \geq up$.

5. We have $sp \geq up$ and $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$. Therefore $\perp :: \Gamma \models (up, \phi') \simeq_U (\mu, \sigma)$ by Lemmas 3 and 4.

6. Since ϵ is immediate $\perp :: \Gamma \models ((\phi\,(sp+1)), up, \phi') \simeq_V (\epsilon, \mu, \sigma)$ hence $\perp :: \Gamma \models (sp+1, (max\ up\, kp), up, \phi') \simeq_Z ((\epsilon :: \zeta), \mu, \sigma)$.

7. We have $sp \geq kp$ and $\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)$. Therefore $\perp :: \Gamma \models (kp, \phi') \simeq_K (\kappa, \sigma)$.

- Assume we have $\Gamma \models ((assign\ \iota\, \pi), up, sp, kp, \phi) \simeq ((assign\ \iota\, \pi), \mu, (\epsilon :: \zeta), \kappa, \sigma)$. We must show that

$$\Gamma \models (\pi, up, sp, kp, \phi') \simeq (\pi, u, (\epsilon :: \zeta), \kappa, \sigma)$$

where $\phi' = \phi[(\phi\ sp)/(\texttt{find}\ \iota\ up\ \phi)]$ and $\sigma' = \sigma[\epsilon/(\mu \downarrow \iota)]$.

We consider each of the conditions for $\simeq$ in turn:

1. By assumption $\pi = \pi$.

2. By assumption Γ is an injection.

3. By assumption $sp \geq kp$.

4. By assumption $sp \geq up$.

5. By assumption $\Gamma \models (up, \phi) \simeq_U (\mu, \sigma)$ and $\Gamma \models ((\phi\ sp), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$ hence $\Gamma \models (up, \phi') \simeq_U (\mu, \sigma')$ by Lemma 6.

6. By assumption, $\Gamma \models (sp, (max\ up\ kp), up, \phi) \simeq_Z (\zeta, \mu, \sigma)$ and $\Gamma \models ((\phi\ sp), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$ hence $\Gamma \models (sp, (max\ up\ kp), up, \phi') \simeq_Z (\zeta, \mu, \sigma')$ by Lemma 6.

7. By assumption $\Gamma \models (kp, \phi) \simeq_K (\kappa, \sigma)$ and $\Gamma \models ((\phi\ sp), up, \phi) \simeq_V (\epsilon, \mu, \sigma)$ hence $\Gamma \models (kp, \phi') \simeq_U (\kappa, \sigma')$ by Lemma 6.

□

Chapter 7

Structuring Back Ends

In chapter 6, we discussed how to represent the runtime structures of the abstract machine on a global stack. To verify that correctness of our representations, we introduced a stack machine that consisted of the same instructions as the original machine but manipulated the stack representations of data. There are several problems with these instructions that need to be addressed:

- The instructions are more powerful than typical machine instructions. For example, the *fetch* instruction follows a static chain pointer to find a lexical reference all in one step.
- The instructions are tree structures. Consider the instruction ($brf\ \pi_0\ \pi_1$). It contains two distinct subtrees π_0 and π_1. In traditional machine architectures, the code store is a linear sequence of instructions accessed through an instruction pointer.
- The only registers used are the special purpose registers *up*, *sp* and *kp*. Contemporary architectures tend to have a rich set of general purpose registers that can be exploited.

In this chapter, we present a register machine based on contemporary RISC architectures. The instructions are much less powerful than those of the stack machine so it will take several register machine instructions to perform the same action that one stack machine instruction performs.

Like the stack machine, the register machine has a linear stack component. The runtime system will manipulate this stack much in the same way the stack machine did, except that contents of some of the stack locations may be kept in registers.

The goal of this chapter is to give a characterization of register machine code that correspond to stack machine code. We parameterize this characterization by assumptions about what stack locations are cached in registers. This characterization of stack and register machine correspondence gives a framework for building compiler back ends and proving them correct (an example back end will be done in chapter 8).

In section 7.1 we present the register machine. We follow this by describing *register assumptions*, which are used to describe register caching strategies, in section 7.2. We use these assumptions in section 7.4 to define a correspondence between stack and register machine states. Finally, we show that if the machines are in corresponding states, the correspondence is preserved by execution of the machines.

7.1 The register machine

The definition of the register machine is shown in Figure 13. There are two distinct states of the register machine - the instruction fetch state and the instruction execution state. The fetch state looks up an instruction to execute and passes control to the execute state. The execute state executes the instruction and passes control back to the fetch state. The register machine consists of three components:

1. γ - A linear code store which is a sequence of register machine instructions. The instructions are modeled after those available in typical RISC machines [61]. The code store cannot be modified during code execution and is left implicit in the machine definition.

2. σ - a stack analogous to the stack machine stack ϕ.

3. δ- a register set. A register can hold the same kind of data that can be held in a stack location. δ is assumed to contain the three registers *ur*, *sr*, and *kr* which will perform the same functions as *up*, *sp*, and *kp* in the stack machine and a register *ir* which points to the next instruction in γ to be executed.

Most of the instructions are straightforward but we give an informal summary of their behavior here.

1. (*movi* $\epsilon\, r$) - move immediate data ϵ into register r.
2. (*mov* $r_0\, r_1$) - move the contents of register r_0 into r_1.
3. (*ld* $r_0\, r_1$) - load the contents of memory pointed to by r_0 into r_1.
4. (*st* $r_0\, r_1$) - store the contents of r_0 into memory at the position pointed to by r_1.
5. (*sw* $r_0\, r_1$) - swap the contents of registers r_0 and r_1.
6. (*inc* $n\, r$) - increment the contents of r by n.
7. (*dec* $n\, r$) - decrement the contents of r by n.
8. (*brf* $n\, r$) - if the contents of register r are false, skip forward n instructions otherwise continue normal execution.
9. (*close* $n\, r\, \nu$) - form a closure of ν arguments with the current environment pointer. The code for the closure is assumed to start at the next instruction and include the next n instructions. The instruction pointer is incremented by n to skip over the code for the closure.
10. (*jmp* r) - go to the instruction pointed to by r.
11. (*rm-unprim*$_i$ r) - perform a unary primitive operation on the contents of register r. The return value is left in r.
12. (*rm-binprim*$_i$ $r_0\, r_1$) - perform a binary primitive operation on the contents of registers r_0 and r_1. The return value is left in register r_1.

Register machine:

Instruction fetch

$$(\sigma, \delta) \implies ((\gamma \downarrow (\delta\ ir)), \sigma, \delta[((\delta\ ir)+1)/\ ir])$$

Instruction execution

$$\begin{array}{rcl}
((movi\ \epsilon\ r), \sigma, \delta) & \implies & (\sigma, \delta[\epsilon/r]) \\
((mov\ r_0\ r_1), \sigma, \delta) & \implies & (\sigma, \delta[(\delta\ r_0)/r_1]) \\
((ld\ r_0\ r_1), \sigma, \delta) & \implies & (\sigma, \delta[(\sigma\,(\delta\ r_0))/r_1]) \\
((st\ r_0\ r_1), \sigma, \delta) & \implies & (\sigma[(\delta\ r_0)/(\delta\ r_1)], \delta) \\
((sw\ r_0\ r_1), \sigma, \delta) & \implies & (\sigma, \delta[(\delta\ r_0)/r_1, (\delta\ r_1)/r_0]) \\
((inc\ n\ r), \sigma, \delta) & \implies & (\sigma, \delta[((\delta\ r)+n)/r]) \\
((dec\ n\ r), \sigma, \delta) & \implies & (\sigma, \delta[((\delta\ r)-n)/r]) \\
((brf\ n\ r), \sigma, \delta) & \implies & (\sigma, \delta[((\delta\ ir)+((\delta\ r) = inB(false) \rightarrow n, 0))/\ ir]) \\
((close\ n\ r\ \nu), \sigma, \delta) & \implies & (\sigma, \delta[inF((\texttt{closure}(\delta\ ur)\ \nu\ ir))/r, ((\delta\ ir)+n)/\ ir]) \\
((jmp\ r), \sigma, \delta) & \implies & (\sigma, \delta[(\delta\ r)/\ ir]) \\
((rm\text{-}unprim_i\ r), \sigma, \delta) & \implies & (\sigma, \delta[(unop_i\,(\delta\ r))/r]) \\
((rm\text{-}binprim_i\ r_0\ r_1), \sigma, \delta) & \implies & (\sigma, \delta[(binop_i\,(\delta\ r_0)\,(\delta\ r_1))/r_1])
\end{array}$$

Figure 13: Actions of the Register Machine

7.2 Register assumptions

We use register assumptions to keep track of which locations on the stack are assumed to be cached in registers. We will use these assumptions to show the correspondence between stack and register machine states. For stack location p in the stack machine, register assumption a can be one of two alternatives:

$$a \ ::= \ m \mid (r, \beta)$$

- The constant m meaning that the location p in the register machine stack contains a value corresponding to the value contained at location p in the stack machine stack and that location is not cached in a register.
- The value (r, β) meaning that register r in the register machine contains a value corresponding to the value contained at location p in the stack machine stack (i.e. we are caching the contents of stack location p in register r). The boolean value β is a

clean bit. If β is true then stack location p in the register machine is guaranteed to contain the same value as register r.

7.2.1 Dynamic register assumptions

Our goal is to use the register machine to simulate the stack machine. During this simulation the register machine's stack, σ, will be identical to the stack machine's stack, ϕ, except for those locations cached in registers. Using a sequence of register assumptions the same size as ϕ, we can dynamically track the stack locations cached in registers[1].

To illustrate, consider the following example of representing the stack machine's stack by the register machine's stack and register set under dynamic register assumptions:

$$\langle (r_0, \mathit{false}), m, (r_3, \mathit{true}), (r_1, \mathit{false}) \rangle$$

Dynamic register assumptions	Stack machine stack ϕ	Register machine stack σ		Register machine register set δ
			r_0	v_0
(r_0, false)	v_0		r_1	v_3
m	v_1	v_1	r_2	unknown
(r_3, true)	v_2	v_2	r_3	v_2
(r_1, false)	v_3	unknown		...

Note that the contents of top location of ϕ is cached in register r_0 so no location exists for it on the register machine stack. While the contents of the bottom location of the stack is cached in register r_1, other values have been pushed on top of that location so space has been reserved for it.

Using dynamic register assumptions we will formally relate states of the stack machine and the register machine in section 7.4.2 using a relation of the form:

[1]The register assumptions are necessary only to prove correspondence between register and stack machine states. Neither machine requires them for execution

$$D \models (\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)$$

which can be read as "under dynamic register assumptions D, register machine state (σ, δ) corresponds to stack machine state (π, up, sp, kp, ϕ)". One major issue in the correspondence between machine states is the proper representation of the stack machine stack by the register machine stack and register set under the dynamic assumptions. The other major issue in the correspondence of machine states is the relationship between the code each machine is to execute. We consider this issue next.

7.2.2 Static register assumptions

In order for the register machine to simulate the stack machine, the code to be executed by the register machine must correspond to the code executed by the stack machine. If stack location p is cached in register r, then for manipulations in the stack machine involving p, the corresponding manipulation in the register machine must use register r. The relationship between register machine code and stack machine code must be parameterized by the register assumptions. However, we would like to generate register machine code from stack machine code statically, without any knowledge of the machine state, so we cannot use the dynamic register assumptions to relate code. Our solution is to use static register assumptions that will predict the structure of the dynamic register assumptions at runtime.

In chapter 6 we saw that there are three distinct types of entities that live on the stack machine stack - the top of stack where local computations are kepts, environment frames, and continuations frames. Our caching strategy will be based upon caching the top of stack and environment frames [2]. Rather than use a fixed register strategy, the notion of correspondence between register and stack machine code will be parameterized by register assumptions about the top of stack and the environment. These assumptions will statically model which locations will have their contents cached in registers at runtime.

[2]Continuation frames can be cached in a manner similar to the approach used for caching environment frames but will not be discussed here.

Stack register assumptions

As shown in the discussion of dynamic register assumptions in section 7.2.1, the top of the stack may be cached in registers. Given that the number of registers is restricted, we can statically track which of the top locations are being cached and in which registers they are being cached by a sequence of register assumptions, S. The idea is that S statically captures the top $\#S$ assumptions that the dynamic register assumptions will have at runtime. Furthermore, S only keeps track of those top locations assumed to be in registers (i.e. S will never contain an assumption m).

For example, if the top two elements of the stack are statically assumed to be cached in registers r_0 and r_1, i.e. $S = (r_0, \beta) :: (r_1, \beta') :: \langle\,\rangle$, then at runtime the dynamic assumptions should be $D = (r_0, \beta) :: (r_1, \beta') :: D'$,

The motivation for this can be shown by the *brf* instruction from the stack machine. This instruction explicitly expects to find its branch value on the top of the stack. The corresponding register machine branch instruction expects to find its branch value in a register. Hence, to simulate the stack machine branch, the top element of the stack must be cached in the register that the register machine is going to branch on.

Environment register assumptions

We will use contours of register assumptions to statically track how the environment will be cached in registers. We do this by associating register assumptions with lexical addresses.

Recall that each lexical contour consists of $n + 1$ contiguous stack locations. The top location is a static chain pointer to the previous contour and the n locations below it contain the values for the current contour. $(U \downarrow \iota)$ returns the register assumption about the static chain about lexical address ι.

At runtime lexical addresses can be turned into stack locations by following the environment pointer *up*. D is an instance of U if for each lexical address in U, D has the same assumption at the corresponding stack location.

As an example, if we have $U = (\langle m, \ldots, (r_5, \mathit{false})\rangle (r_1, \mathit{true}) \cdot \langle m, (r_7, \mathit{true}), m, m\rangle)$ then at runtime D should look like:

Dynamic register assumptions D		Stack machine stack ϕ	
$\vdots$		$\vdots$	
m	up :	(`closure` $2\ up'\ \pi$)	lexical address $(0,0)$
(r_5, false)		v_1	lexical address $(0,1)$
(r_1, true)		v_2	lexical address $(0,2)$
$\vdots$		$\vdots$	
m	up' :	(`closure` $3\ -1\ \pi'$)	lexical address $(1,0)$
(r_7, true)		v'_1	lexical address $(1,1)$
m		v'_2	lexical address $(1,2)$
m		v'_3	lexical address $(1,3)$
$\vdots$		$\vdots$	

7.3 Relating register and stack machine code

Given the static register assumptions we can now set out to formally relate register machine code to stack machine code.

7.3.1 Procedure call/return protocols

The standard technique for passing values to and from procedures is to use some kind of register conventions. Typical conventions would say something like "all parameters are passed in registers r_1 through r_n" or "results of procedure calls are always placed in register r_0". There are numerous valid conventions and it is up to the language implementor to decide which suits his needs most. As our relationship between stack and register machine code is to be parameterized by register assumptions, we must have some knowledge of the register assumptions before and after procedure calls. Rather than lock into any one strategy, we will abstract the necessary strategies with functions that will return the required stack and environment register assumptions (as a pair).

- (`call` $\nu\, U$) $= (S', U')$

  For a call to a procedure of ν arguments from an environment with assumptions U, two values are returned:

  1. the stack state S' telling how to pass the arguments
  2. the environment state U' telling what state the caller's environment should be in

- (`tailcall` $\nu\, U$) $= (S', U')$

  For a call to a procedure of ν arguments from an environment with assumptions U, two values are returned:

  1. the stack state S' telling how to pass the arguments
  2. the environment state U' telling what state the caller's environment should be in

- (`entry` $\nu\, U$) $= (S', U')$

  For a called procedure of ν arguments closed in an environment with assumptions U, two values are returned:

  1. the stack state at the entry of the procedure
  2. the environment state U' telling how the body of the procedure expects to find registers cached when it is entered

- (`exit` U) $= (S', U')$

  For the return point of a procedure called from an environment with assumptions U, two values are returned:

  1. the stack state telling how to return values

2. the environment state U' telling how the environment should be saved before returning

- (`return` U) $= (S', U')$

  For the point to where a procedure will return with environment with assumptions U, two values are returned:

  1. the stack state telling how values are returned
  2. the environment state U' telling how the environment is expected to look after the procedure returns

While the strategies are left up to the back end implementation, certain restrictions must be enforced for them to work properly. For example, if procedures always return values in register r_0 then the return point from procedure calls should expect to find the result in register r_0. Before formalizing these restrictions, we will first introduce some new terminology to aid in the the discussion.

For environment register assumptions, we can produce a contour of register assumptions with the same shape but with all of the assumptions as m by the following:

$$(\texttt{mem}\ U) = \begin{cases} \text{if } U = \langle\,\rangle \text{ then } U \\ \text{if } U = (\langle a_1, \ldots, a_n\rangle \cdot U') \text{ then } (\langle m_1, \ldots, m_n\rangle \cdot (\texttt{mem}\ U')) \end{cases}$$

Environment register assumptions track environments. Just as environments can be inferior to other environments, environment assumptions can be inferior to other environment assumptions.

Definition 5 *A contour of register assumptions U is* inferior *to U' if* $(\texttt{mem}\ U) \subset_C (\texttt{mem}\ U')$.

We can now start our discussion of the restrictions on the protocols. The protocols `call` and `entry` relate how parameters are passed from a calling point to the entry point of a procedure. Because of stack discipline, the environment of the procedure (except for the arguments) is inferior to the environment of calling point. When the call is made, the calling point should have the registers set up the way the entry point of the procedure expects to see them and should have the rest of the environment should be in memory (including the parts of the calling environment superior to the procedure's environment). Recall that arguments are passed to procedures on the stack. These values form the topmost contour of the procedure's environment. For our protocols, this means that the calling point should have the top of the stack cached in the registers that the procedure expects to see its topmost environment contour cached.

Given environment assumptions U' and inferior environment assumptions U, we can test that the superior part of U' is in memory and the rest of the assumptions match with the following test.

$$(U \subset^m_C U') = \begin{cases} \text{if } U = U' \text{ then } \mathit{true} \\ \text{if } U' = (\langle m_1, \ldots, m_n \rangle \cdot U'') \text{ then } (U \subset^m_C U'') \\ \text{otherwise } \mathit{false} \end{cases}$$

We can now formalize the restrictions on the `call` and `entry` protocols:

Definition 6 *Procedure calling protocols* `call` *and* `entry` *are* valid protocols *if for environment assumptions* U_0 *with inferior environment assumptions* U'_0*,* $(S, U) = (\texttt{call}\ \nu\, U_0)$ *and* $(\langle\,\rangle, (S' \cdot U')) = (\texttt{entry}\ \nu\, U'_0)$ *then* $S' = S$ *and* $U' \subset^m_C U$*.*

The restriction on `tailcall` and `entry` are quite similar except that in a tail call, the topmost environment contour disappears. Since it is never used again, we need not save the contour so we add the additional caveat that the topmost contour of the calling procedure need not be in memory.

Definition 7 *Procedure calling protocols* `tailcall` *and* `entry` *are* valid protocols *if for environment assumptions* $(C_0 \cdot U_0)$ *with inferior environment assumptions* U'_0*,* $(S, (C \cdot U)) = (\texttt{call}\ \nu\, (C_0 \cdot U_0))$ *and* $(\langle\,\rangle, (S' \cdot U')) = (\texttt{entry}\ \nu\, U'_0)$ *then* $S' = S$ *and* $U' \subset^m_C U$*.*

Similar reasoning is used when a procedure returns to its calling point. The value being returned must be passed in the register the return point expects it in and the environment must cached in the registers the return point expects to see its environment cached in.

Definition 8 *Procedure return protocols* `exit` *and* `return` *are* valid protocols *if for environment assumptions* U'_0 *with inferior environment assumptions* U_0, $(S, U) = (\texttt{exit}\,(C \cdot U_0))$ *and* $(S', U') = (\texttt{return}\ U'_0)$ *then* $S = S'$ *and* $U \subset^m_C U'$.

7.3.2 Derived instructions

The registers *ur*, *sr*, *kr*, and *ir* do not cache the contents of the stack but rather are used for other purposes. As our register allocation is characterized in terms of caching parts of the stack, our formal system does not deal with these special registers[3]. However, their uses are restricted to certain well-defined instances which we've encapsulate in the "derived instructions"[4] shown in Figure 14. These instructions allow us to reason explicitly about the uses of the special registers. Their behavior is defined in terms of the usual register machine instructions but, for clarity, we will summarize their behavior:

1. (*push* r) - increment the stack pointer and store the contents of r on top of the stack.
2. (*pop* r) - put the contents of the top of stack into r and decrement the stack pointer.
3. (*ld* $r_0(n)\, r_1$) - load the contents of the stack location pointed to by r_0 offset by n into r_1.
4. (*st* $r_0\, r_1(n)$) - store the contents of r_0 into the stack location pointed to by r_1 offset by n.
5. (*goto* n) - skip forward n instructions.
6. (*deref* $r(n)$) - load the contents of the stack location pointed to by r offset by n into r.
7. (*catch* $n\, r$) - push a continuation on the stack and update the continuation pointer. The code for the continuation occurs n instructions forward from the current one. r is used as an intermediary and contains garbage afterwards.

[3]Our techniques can be extended to cover the use of registers. We hope to report on this later.
[4]The reader may consider derived instructions as macros, if so inclined.

$$(push\ r) = \begin{cases} (inc\ sr) \\ (st\ r\ sr) \end{cases} \qquad (pop\ r) = \begin{cases} (ld\ sr\ r) \\ (dec\ sr) \end{cases}$$

$$(ld\ r_0(n)\, r_1) = \begin{cases} (dec\ n\, r_0) \\ (ld\ r_0\, r_1) \\ (inc\ n\, r_0) \end{cases} \qquad (st\ r_0\, r_1(n)) = \begin{cases} (dec\ n\, r_1) \\ (st\ r_0\, r_1) \\ (inc\ n\, r_1) \end{cases}$$

$$(goto\ n) = (inc\ n\ ir) \qquad (deref\, r(n)) = \begin{cases} (dec\ n\, r) \\ (ld\ r\, r) \end{cases}$$

$$(catch\ n\, r) = \begin{cases} (push\ kr) \\ (push\ ur) \\ (mov\ ir\ r) \\ (mov\ sr\ kr) \\ (inc\ r\, n+1) \\ (push\ r) \end{cases} \qquad (rts) = \begin{cases} (mov\ kr\ sr) \\ (pop\ ur) \\ (pop\ kr) \\ (pop\ ir) \end{cases}$$

Figure 14: Derived Instruction Sequences

8. (rts) - the steps to return from a procedure:

   (a) move the current continuation pointer into the stack pointer (effectively pops the stack to the current continuation)

   (b) pop the saved environment pointer into the environment register

   (c) pop the saved continuation pointer into the continuation register

   (d) pop the saved instruction pointer into the instruction register. (causes a "jump" to the saved return address)

7.3.3 The formal system

We are now able to define a static correspondence between stack machine code and register machine code. The notation

$$(S, U) \models \gamma \simeq_P \pi$$

can be read as "under stack assumptions S and environment assumptions U, register machine code γ corresponds to stack machine code π".

All of the registers we will use are elements of $R = \{\delta\text{'s registers}\} - \{ur, sr, kr, ir\}$. In addition, given a sequence of register assumptions (such as D and S), we can find the registers not used by the assumptions:

$$R/a^* = \begin{cases} \text{if } a^* = \langle\,\rangle \text{ then } R \\ \text{if } a^* = m :: a^{*\prime} \text{ then } R/a^{*\prime} \\ \text{if } a^* = (r, \beta) :: a^{*\prime} \text{ then } (R - \{r\})/a^{*\prime} \end{cases}$$

noting that an error occurs if a^* contains a register not in R. Similarly, we can also collect the free registers from a contour of register assumptions:

$$R/c = \begin{cases} \text{if } c = \langle\,\rangle \text{ then } R \\ \text{if } c = (a^* \cdot c') \text{ then } (R/a^*)/c' \end{cases}$$

Direct correspondence

The following set of inference rules formalizes the correspondence between stack and register machine code.

The *goto* instruction in the register machine skips forward n instructions. Hence if we have a correspondence between some register and stack machine code, we can append a *goto* followed by n instructions to the register machine code and the code still corresponds.

$$\frac{(S, U) \models \gamma_0 \simeq_P \pi_0 \quad \#\gamma_1 = n}{(S, U) \models (\mathit{goto}\ n) :: \gamma_1 \S \gamma_0 \simeq_P \pi_0}$$

The stack machine *goto* instruction just executes its contained code. If some stack and register machine code correspond we can wrap the stack machine code in a *goto* and the code still corresponds.

$$\frac{(S, U) \models \gamma_0 \simeq_P \pi_0}{(S, U) \models \gamma_0 \simeq_P (\mathit{goto}\ \pi_0)}$$

The stack machine *constant* instruction pushes immediate data onto the stack. We simulate this by moving the same data into a free register that now caches the top of the stack.

$$\frac{r \notin S \quad r \notin U \qquad (r, \mathit{false})::(S,U) \models \gamma_0 \simeq_P \pi_0}{(S,U) \models (\mathit{movi}\ \epsilon\ r)::\gamma_0 \simeq_P (\mathit{constant}\ \epsilon\ \pi_0)}$$

The *fetch* instruction in the stack machine looks up the contents of the stack at the specified lexical address and pushes that value on top of the stack. We simulate this by caching that lexical address in a register and moving its contents to a free register. The free register now caches the top of the stack.

$$\frac{r_1 \notin S \quad r_1 \notin U \qquad U \downarrow \iota = (r_0, \beta) \quad (r_1, \mathit{false})::(S,U) \models \gamma_0 \simeq_P \pi_0}{(S,U) \models (\mathit{mov}\ r_0\ r_1)::\gamma_0 \simeq_P (\mathit{fetch}\ \iota\ \pi_0)}$$

The *assign* stack machine instruction takes the value on top of the stack and stores it at the specified lexical address. We simulate this by caching the top of the stack in a register, caching the lexical address in a register and moving the contents of the first register to the second. The overwritten register no longer agrees with the stack location it represents so the clean bit is set to *false*.

$$\frac{S = (r_0, \beta)::S' \quad U \downarrow \iota = (r_1, \beta) \qquad (S, U[(r_1, \mathit{false})/i]) \models \gamma_0 \simeq_P \pi_0}{(S,U) \models (\mathit{mov}\ r_0\ r_1)::\gamma_0 \simeq_P (\mathit{assign}\ \iota\ \pi_0)}$$

The stack machine *brf* instruction pops the value from the top of the stack and branches on its value. We simulate this by caching the top of the stack in a register and deallocating it after we've tested its value. If the value is not true, n instructions are skipped to a point where the sequence must to the false branch of the *brf*. If the value is true, execution continues ar the next instruction. The sequence at this point must correspond to the *brf*'s true branch.

$$\frac{S = (r, \beta)::S' \quad \#\gamma_0 = n \qquad (S', U) \models \gamma_0 \S \gamma_1 \simeq_P \pi_0 \quad (S', U) \models \gamma_1 \simeq_P \pi_1}{(S,U) \models (\mathit{brf}\ n\ r)::\gamma_0 \S \gamma_1 \simeq_P (\mathit{brf}\ \pi_0\ \pi_1)}$$

$$\frac{r \notin S \quad r \notin U \quad \#\gamma_0 = n \qquad (\texttt{entry}\ \nu\, U) \models \gamma_0 \simeq_P \pi_0 \quad (r, \mathit{false}) :: S, U \models \gamma_1 \simeq_P \pi_1}{(S, U) \models (\mathit{close}\ n\ r\ \nu) :: \gamma_0 \S \gamma_1 \simeq_P (\mathit{close}\ \nu\, \pi_0\, \pi_1)}$$

In the stack machine, the *rts* instruction returns from a procedure by restoring the current continuation. Once the stack and environment assumptions match our return protocol, the derived instruction *rts* will perform the same action.

$$\frac{S, U = (\texttt{exit}\ U)}{(S, U) \models (\mathit{rts}) \simeq_P (\mathit{rts})}$$

The *jmp* instruction on the stack machine calls a procedure of ν arguments. Once the stack and environment assumptions match our call protocol, the *jmp* instruction will perform the same action.

$$\frac{r \notin S \quad r \notin U \qquad (S, U) = (\texttt{tailcall}\ \nu\, U)}{(S, U) \models (\mathit{jmp}\ r) \simeq_P (\mathit{jmp}\ \nu)}$$

The *jsr* instruction on the stack machine calls a procedure of ν arguments. Once the stack and environment assumptions match our call protocol, the *jmp* instruction will perform the same action.

$$\frac{r \notin S \quad r \notin U \qquad S, U = (\texttt{call}\ \nu\, U)}{(S, U) \models (\mathit{jmp}\ r) \simeq_P (\mathit{jsr}\ \nu)}$$

$$\frac{S = \langle\,\rangle \quad \#\gamma_0 = n \qquad (\texttt{return}\ U) \models \gamma_0 \simeq_P \pi_0 \quad (S, U) \models \gamma_1 \simeq_P \pi_1}{(S, U) \models (\mathit{catch}\ n) :: \gamma_0 \S \gamma_1 \simeq_P (\mathit{catch}\ \pi_0\, \pi_1)}$$

We don't explicitly test for errors so the stack machine *test-arg* instruction just executes its contained code. If some stack and register machine code correspond, we can wrap the stack machine code in a *test-arg* and the code still corresponds.

$$\frac{(S, U) \models \gamma_0 \simeq_P \pi_0}{(S, U) \models \gamma_0 \simeq_P (\mathit{test\text{-}arg}\ \pi_0)}$$

The stack machine *pop* instruction pops a value off the stack. We simulate this by caching the top of the stack in a register and deallocating the register.

$$\frac{S = (r, \beta) :: S' \quad (S', U) \models \gamma_0 \simeq_P \pi_0}{(S, U) \models \gamma_0 \simeq_P (\mathit{pop}\ \pi_0)}$$

The stack machine unary operators pop the top element from the stack and push a result. We simulate this in the register machine by caching the top of the stack in a register. The corresponding register machine unary operator takes the contents of the register and overwrites it with the result.

$$\frac{\begin{array}{c} S = (r, \beta) :: S' \\ (r, \mathit{false}) :: (S', U) \models \gamma_0 \simeq_P \pi_0 \end{array}}{(S, U) \models (\mathit{rm\text{-}unprim}_i\ r) :: \gamma_0 \simeq_P (\mathit{sm\text{-}unprim}_i\ \pi_0)}$$

The stack machine binary operators pop the top two element from the stack and push a result. We simulate this in the register machine by caching the top two elements of the stack in a register. The corresponding register machine binary operator takes the contents of the two registers and overwrites the second one with the result.

$$\frac{\begin{array}{c} S = (r_1, \beta) :: (r_0, \beta) :: S' \\ (r_1, \mathit{false}) :: (S', U) \models \gamma_0 \simeq_P \pi_0 \end{array}}{(S, U) \models (\mathit{rm\text{-}binprim}_i\ r_0\, r_1) :: \gamma_0 \simeq_P (\mathit{sm\text{-}binprim}_i\ \pi_0)}$$

Stack manipulations

The following inference rules tell us how we can shuffle the top of the stack under the current register assumptions.

To increase the number of elements being cached, we can pop the current top of the register machine's stack into an unused register and add that to the *back* of S. This is because the things currently in S are caching the elements above the current top of stack.

$$\frac{\begin{array}{c} r \notin S \quad r \notin U \\ (S :: (r, \mathit{true}), U) \models \gamma \simeq_P \pi \end{array}}{(S, U) \models (\mathit{pop}\ r) :: \gamma \simeq_P \pi}$$

Similarly, to decrease the number of elements being cached, we can push the register currently caching the last element of S.

$$\frac{S = S' :: (r, \mathit{false}) \qquad (S', U) \models \gamma \simeq_P \pi}{(S, U) \models (\mathit{push}\ r) :: \gamma \simeq_P \pi}$$

Environment manipulations

These rules tell us how we can use registers to follow static chain pointers and reference parts of the environment.

Since ur always points to the top of the current lexical contour, we can use ur to load the topmost static chain pointer into a register.

$$\frac{r \notin S \quad r \notin U \qquad (S, U[(r, \mathit{true})/(0,0)]) \models \gamma \simeq_P \pi}{(S, U) \models (\mathit{ld}\ ur\ r) :: \gamma \simeq_P \pi}$$

If r contains a static chain pointer to lexical contour $d+1$ then we can load the oth element of that contour into r through a *deref*. Furthermore, r no longer caches the static chain pointer but does cache the $(d+1, o)$ element of the environment.

$$\frac{r \notin S \quad U \downarrow (d, 0) = (r, \mathit{true}) \qquad (S, (U[m/(d,0)])[(r, \mathit{true})/(d+1, o)]) \models \gamma \simeq_P \pi}{(S, U) \models (\mathit{deref}\ r(o)) :: \gamma \simeq_P \pi}$$

If r_0 contains a static chain pointer to lexical contour $d+1$ then we can load the oth element of that contour into an unused register, r_1, through a *ld*. Furthermore, the environment now has its $(d+1, o)$ element cached in r_1.

$$\frac{r_1 \notin S \quad r_1 \notin U \quad U \downarrow (d, 0) = (r_0, \beta) \qquad (S, U[(r_1, \mathit{true})/(d+1, o)]) \models \gamma \simeq_P \pi}{(S, U) \models (\mathit{ld}\ r_0(o)\ r_1) :: \gamma \simeq_P \pi}$$

If r_1 contains a static chain pointer to lexical contour $d+1$ then we can store the contents of register r_0 into oth element of that contour through a *st*. Furthermore, the environment now has its $(d+1, o)$ element cached in r_0.

$$\frac{U \downarrow (d,0) = (r_1, \beta) \qquad (S, U[(r_0, \mathit{true})/(d+1, o)]) \models \gamma \simeq_P \pi}{(S,U) \models (st\ r_0\, r_1(o)) :: \gamma \simeq_P \pi}$$

Finally, if a register is caching a lexical address and the register's contents are clean, we can implicitly deallocate the register.

$$\frac{U \downarrow (d,o) = (r_1, \mathit{true}) \qquad (S, U[m/(d,o)]) \models \gamma \simeq_P \pi}{(S,U) \models \gamma \simeq_P \pi}$$

7.4 Simulating the stack machine

We now show that if some register and stack machine code correspond and the dynamic state of the machine conforms to the static register assumptions then the register machine simulates the stack machine.

7.4.1 Extracting static register assumptions from dynamic ones

Recall from section 7.2.1 that the dynamic register assumptions tell us about the representation of every location in the stack machine's stack ϕ. In order to use this information to check the correspondence of the machines' code, we have to extract the stack and environment assumptions for which the dynamic assumptions are an instance. In Theorem 3 we will see that if the stack machine's stack is represented properly in the register machine's stack and register set under the dynamic assumptions and the stack and register machine code correspond under the extracted static assumptions, then the machines will stay in related state throughout their execution. Hence if the machines are started in related states then upon termination they will still be in related states (i.e. they will return corresponding answers).

Extracting stack assumptions

Given the number of positions on the top of the stack assumed to be cached in registers, extracting the static stack assumptions is a matter of copying that number of elements from the top of D.

$$\texttt{extract}_S(D\, n) = \langle (D \downarrow 0), \ldots, (D \downarrow (n-1)) \rangle$$

Extracting environment assumptions

Extracting environment assumptions is slightly harder as we have to follow the static chain pointer and return a contour of assumptions. Given an environment pointer up and the stack ϕ, if up points to the initial environment we return an empty contour (no assumptions). Otherwise we must:

1. extract the current lexical contour into a sequence of register assumptions
2. extract the rest of the environment into a contour of register assumptions by following the static chain pointer
3. add the sequence of assumption to the front of the contour of assumptions

which we can formally define as:

$$\texttt{extract}_U(D\ up\ \phi) = \begin{cases} \text{if } p = -1 \text{ then } \langle\,\rangle \\ \text{if } \phi(up) = (\texttt{closure}\ \nu\ up\ \pi) \text{ then} \\ \quad (\langle (D \uparrow up), \ldots, (D \uparrow (up - \nu)) \rangle \cdot \texttt{extract}_U(D\, \phi(up).up\, \phi)) \end{cases}$$

7.4.2 Representations

Now the correspondence between the register machine and stack machine can be stated.

Values

A register machine value corresponds to a stack machine value if either they are the same value or if they are both closures whose code corresponds under the parameter passing protocols.

$$D \models (v, \sigma, \delta) \simeq_V (v, \phi)$$

$$\frac{v = (\texttt{closure}\ \nu\ ur\ ir) \quad v' = (\texttt{closure}\ \nu\ up\ \pi) \quad ur = up \qquad (\texttt{entry}\ \nu\ \texttt{extract}_U(D\ up\ \phi)) \models (\gamma \dagger (ir - 1)) \simeq_P \pi}{D \models (v, \sigma, \delta) \simeq_V (v', \phi)}$$

Locations

A location on the stack in the stack machine is represented properly if

- The location is not cached in a register and the same location in the register machine contains a corresponding value.
- The location is cached in a register and the register contains a corresponding value.

For cases when the register assumption about location p is (r, true) we must also verify that r and location p in the register machine contain the same value.

$$\frac{(\phi\, p) = v' \qquad (D \uparrow p) = m \quad (\sigma\, p) = v \qquad D \models (v, \sigma, \delta) \simeq_V (v', \phi)}{D \models (p, \sigma, \delta) \simeq_p (p, \phi)}$$

$$\frac{(\phi\, p) = v \qquad (D \uparrow p) = (r, \beta) \quad (\delta\, r) = v \quad \beta \rightarrow (\delta\, r) = (\sigma\, p)}{D \models (p, \sigma, \delta) \simeq_p (p, \phi)}$$

Sequences of locations

A sequence of stack locations is represented properly if each individual location is properly represented.

$$\frac{p_l > p_u}{D \models (\sigma, \delta) \simeq_{p_l, p_u} \phi}$$

$$\frac{D \models (p_l, \sigma, \delta) \simeq_V (p_l, \phi) \qquad D \models (\sigma, \delta) \simeq_{(p_l+1), p_u} \phi}{D \models (\sigma, \delta) \simeq_{p_l, p_u} \phi}$$

Machine states

Finally, machine states correspond if:

- the entire stack is properly represented
- the continuation pointers are the same
- the environment pointers are the same
- the stack pointers differ by the number of words on the top being cached in registers.
- The stack and environment assumptions extracted from the dynamic assumptions give a correspondence between the code to be executed on both machines.

$$\frac{\begin{array}{c} \#D = sp \quad D \models (\sigma, \delta) \simeq_{1,sp} \phi \\ (\delta\ ur) = up \quad (\delta\ kr) = kp \quad n = sp - (\delta\ sr) \\ R/D = R/\, \mathtt{extract}_S(D\, n) \cap R/\, \mathtt{extract}_U(D\ up\, \phi) \\ (\mathtt{extract}_S(D\, n), \mathtt{extract}_U(D\ up\ \phi)) \models \gamma \dagger ((\delta\ ir) - 1) \simeq \pi \end{array}}{D \models (\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)}$$

which can be extended to execute states by noting that the fetch state transforms into an execute state by simultaneously looking up the instruction pointed to by the instruction register ir and incrementing ir:

$$\frac{D \models (\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)}{D \models (\gamma \downarrow (\delta\ ir), \sigma, \delta[(\delta\ ir) + 1/\ ir]) \simeq (\pi, up, sp, kp, \phi)}$$

7.5 Correctness of simulation

Using the formalizations of the representations, we can now prove that the register machine will simulate the stack machine. The first step towards this proof is to show that at runtime, our static assumptions about lexical addresses correspond to the dynamic assumption about the actual locations associated with lexical addresses.

Lemma 7 *If* $D \models (\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)$ *and* $(\texttt{extract}_U(D\ up\ \phi) \downarrow \iota) = a$ *then* $(D \uparrow (\texttt{find}\ \iota\ up\ \phi)) = a$.

Proof: By definition of $\texttt{extract}_U$ and $\texttt{find}$. $\Box$.

Definition 9 *We say register machine state* (σ, δ) corresponds *to a stack machine state* (π, up, sp, kp, ϕ) *(written* $(\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)$*) if there exists a dynamic stack assumption* D *such that* $D \models (\sigma, \delta) \simeq (\pi, up, sp, kp, \phi)$.

Now we can show the main theorem, which asserts that the register machine correctly simulates the stack machine:

Theorem 3 *Let* S_1 *and* S_2 *be states of the stack machine and* R_1 *be a state of the register machine. If* $R_1 \simeq S_1$, $S_1 \Longrightarrow S_2$, *then there exists* R_2 *such that* $R_1 \Longrightarrow^* R_2$ *and* $R_2 \simeq S_2$.

Proof: By analysis of each instruction. Here we show *fetch*.

Assume we have

$$D \models ((mov\ r_0\, r_1\,), \sigma, \delta) \simeq ((fetch\ \iota\, \pi), up, sp, kp, \phi)$$

We must show that

$$D' \models (\sigma, \delta') \simeq (\pi, up, sp{+}1, kp, \phi')$$

$$\text{where } \begin{cases} D' = (r_1, \mathit{false}) :: D \\ \delta' = \delta[(\delta\, r_0)/r_1] \\ \phi' = \phi[(\phi\,(\texttt{find}\ \iota\ up\,\phi))/(sp{+}1)] \end{cases}$$

We consider each of the conditions for $\simeq$ in turn:

Let

$$n = sp - (\delta\ sr)$$

$$S = \mathtt{extract}_S(D\, n) \qquad U = \mathtt{extract}_U(D\ up\, \phi)$$

$$S' = \mathtt{extract}_S(D'\, n+1) \qquad U' = \mathtt{extract}_U(D'\ up\ \phi')$$

- $(\delta\ ur) = up$.
- $(\delta\ kr) = kp$.
- $n' = sp + 1 - (\delta\ sr) = n + 1$
- $\#D = sp \rightarrow \#D' = sp + 1$.
- $n' = n + 1$ thus $S' = (r_1, \mathit{false}) :: S.up \leq sp$ hence $U = U'.r_1 \notin S$ and $r_1 \notin U$ and by assumption $(S, U) \models (mov\ r_0\, r_1) :: (\gamma \dagger (\delta\ ir) - 1) \simeq_P (fetch\ \iota\, \pi)$ hence $S', U' \models (\gamma \dagger (\delta\ ir) - 1) \simeq_P \pi$.
- $R/D = R/S \cap R/U$ hence $R/D' = R/S' \cap R/U'$.
- $D \models (\sigma, \delta) \simeq_{1,sp} \phi$ and $up \leq sp$ hence $D \models ((\mathtt{find}\ \iota\ up\, \phi), \sigma, \delta') \simeq_P ((\mathtt{find}\ \iota\ up\, \phi), \phi)$.

  By $\simeq_P$, $U(l) = (r_0, \beta)$ thus by Lemma 7, $(D \uparrow (\mathtt{find}\ \iota\ up\, \phi)) = (r_0, \beta)$. Hence $D' \models (sp+1, \sigma, \delta') \simeq_P (sp+1, \phi')$

  Thus $D' \models (\sigma, \delta') \simeq_{1,sp+1} \phi'$.

The other instructions are similar.

$\square$

Chapter 8

A Simple Back End

In this chapter, we present and verify a simple back end. The job of the back end is to translate stack machine code into an equivalent sequence of register machine instructions as specified in section 7.3. Theorem3 then guarantees that the stack machine, and the register machine, started with the original stack machine code and the translated code, will always stay in equivalent states. There are two major concerns in this process:

- Code linearization - register machine code is inherently linear, whereas stack machine code is tree structured. The situation is complicated in presence of *brf*s, where stack machine code is actually a dag rather than a tree. Code duplication is an unacceptable method of linearization so the issue has to be dealt with carefully.

- Register allocation - for register allocation, the back end is parameterized by stack and environment register assumptions. Our intent is to produce a sequence of register machine instructions from the stack machine code under the register assumptions such that the local computations and portions of the environment will be kept in registers. This approach will allow us to formalize the correctness of our code generator.

8.1 Notation

In this chapter we use a $\texttt{let}^*$ notation in the metalanguage as an abbreviation for nested `let` statements. Formally, we mean:

$$(\texttt{let}^*\ \ i = e\ i' = e' \ldots\ \texttt{in}\ b) \equiv (\texttt{let}\ \ i = e\ \ \texttt{in}\ \texttt{let}^*\, i' = e' \ldots\ \texttt{in}\ b)$$

$$(\texttt{let}^*\ \ i = e\ \ \texttt{in}\ b) \equiv (\texttt{let}\ \ i = e\ \ \texttt{in}\ b)$$

8.2 Register usage conventions

We divide the register machine's register set, R, into two distinct sets, R_S and R_U. Registers from R_S, denoted s_i, will be used to cache the top of stack and registers from R_U, denoted u_i, will be used to cache the environment.

$$R = R_S \cup R_U \ \text{ and } \ R_S \cap R_U = \emptyset$$

We will pass parameters in registers $u_1 \ldots u_n$ and the static chain pointer will be in register u_0. Space for the parameters will be allocated on the stack. For simplicity, we will assume that for every procedure there are enough registers to pass its parameters[1]. Furthermore, we will return values from procedure calls in register s_0.

We formalize our register conventions in Table 1 by giving a definition of the call/return protocols discussed in section 7.3.1. Our definition is aided by the function $(\texttt{mem}\ U)$ from the last chapter. Given environment assumptions U, it returns environment assumptions similar to U except that all elements are assumed to be in memory:

$$(\texttt{mem}\ U) = \begin{cases} \text{if } U = \langle\,\rangle \text{ then } U \\ \text{if } U = (\langle a_1, \ldots, a_n\rangle \cdot U') \text{ then } (\langle m_1, \ldots, m_n\rangle \cdot (\texttt{mem}\ U')) \end{cases}$$

Lemma 8 *The protocols specified in Table 1 are valid protocols as specified in section 7.3.1.*

Proof: by definition. $\square$

[1] We could just pass extra parameters directly on the stack, as is often done in practice. As this has been discussed thoroughly elsewhere [2, 10, 31], we will spare the reader the details.

$$
\begin{array}{rcl}
(\texttt{call}\ \nu\, U) & = & (\langle (u_0, \mathit{false}), \ldots, (u_\nu, \mathit{false})\rangle, (\texttt{mem}\ U)) \\
(\texttt{tailcall}\ \nu\, (C \cdot U)) & = & (\langle (u_0, \mathit{false}), \ldots, (u_\nu, \mathit{false})\rangle, (\texttt{mem}\ U)) \\
(\texttt{entry}\ \nu\, U) & = & (\langle\ \rangle, (\langle (u_0, \mathit{false}), \ldots, (u_\nu, \mathit{false})\rangle \cdot (\texttt{mem}\ U))) \\
(\texttt{exit}\, (C \cdot U)) & = & (\langle (s_0, \mathit{false})\rangle, (C \cdot (\texttt{mem}\ U))) \\
(\texttt{return}\ U) & = & (\langle (s_0, \mathit{false})\rangle, (\texttt{mem}\ U))
\end{array}
$$

Table 1: Procedure call/return protocols

8.3 Stack machine program structure

The `if` expression causes a split in the program graph that rejoins later at a common point. This common point is the compiler's code continuation, π, which is the code to follow the `if` expression. To identify these join points, the compiler inserts *goto* instructions:

$$\mathcal{CE}[\![(\texttt{if}\ \texttt{E}_0\ \texttt{E}_1\ \texttt{E}_2)]\!]\tau\pi = \mathcal{CE}[\![\texttt{E}_0]\!]\tau(\mathit{brf}(\mathcal{CE}[\![\texttt{E}_1]\!]\tau(\underline{\mathit{goto}\,\pi}))\,(\mathcal{CE}[\![\texttt{E}_2]\!]\tau(\underline{\mathit{goto}\,\pi})))$$

To find the join point of a particular branch, you follow the code graph until you find an unmatched *goto* (similar to matching nested parenthesis). The `if` expression is the only expression for which the compiler generates *goto* instructions so no extraneous *goto*'s should occur in the code. This restriction is not inherent in the definition of stack machine code so we must formalize this notion of legal stack machine code.

We begin by identifying the leaves of the tree structured, stack machine code. These are the instructions that contain no code within themselves.

Definition 10 *The stack machine instructions rts, jmp, and jsr form the* leaves *of stack machine code.*

We now give a method for traversing stack machine code to return the join point contained within the first unmatched *goto*. If we run into a leaf before finding an unmatched *goto*, the leaf is returned.

Definition 11 *The* join point *of* π,(`join` π)*, is specified in Table 2.*

Using these definitions, we can now define legal stack machine code. Our main concern is to verify that for every *brf* instruction, its two branches rejoin at a common join point.

Definition 12 *π is* legal *if* (`legal` π) = *true as specified in Table 2.*

Lemma 9 *If* (`legal` π) *and* (`join` π) = π' *then* (`legal` π').

Proof: Structural induction on π. □

We conclude this section by stating that the compiler generates legal code. This should not be too surprising as our original intention was to characterize the code generated by the compiler.

Theorem 4 *If π is legal then*

- $\mathcal{CE}[\![\mathtt{E}]\!]\tau\pi$ *is legal*
- $\mathcal{CT}[\![\mathtt{E}]\!]\tau$ *is legal*
- $\mathcal{CE}^*[\![\mathtt{E}^*]\!]\tau\pi$ *is legal*
- $\mathcal{CA}^*[\![\mathtt{E}^*]\!]\tau\pi$ *is legal*
- $\mathcal{CS}^*[\![\mathtt{E}^*]\!]\tau\pi$ *is legal*

Proof: Structural induction on expressions. □

8.4 Reasonable register assumptions

In the last chapter we showed how to relate stack machine code to register machine code using register assumptions. The structure of the stack machine code imposes certain restrictions upon the register assumptions that can be used in this relation. For example, the stack machine instruction *fetch* contains an explicit reference to a lexical address. If the environment assumptions are not defined for that lexical address, we cannot relate the

π	(`join` π)	(`legal` π)
$(goto\ \pi_0)$	π_0	(`legal` π_0)
$(constant\ \epsilon\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(fetch\ \iota\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(assign\ \iota\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(brf\ \pi_0\ \pi_1)$	(`join` (`join` π_0))	(`legal` π_0) and (`legal` π_1) and (`join` π_0) $=$ (`join` π_1)
$(brfng\ \pi_0\ \pi_1)$	(`join` π_1)	(`legal` π_0) and (`legal` π_1)
$(close\ \nu\ \pi_0\ \pi_1)$	(`join` π_0)	(`legal` π_0) and (`legal` π_1)
(rts)	(rts)	*true*
$(catch\ \pi_0\ \pi_1)$	(`join` π_0)	(`legal` π_0) and (`legal` π_1)
$(jmp\ \nu)$	$(jmp\ \nu)$	*true*
$(jsr\ \nu)$	$(jsr\ \nu)$	*true*
$(test\text{-}arg\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(pop\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(sm\text{-}unprim_i\ \pi_0)$	(`join` π_0)	(`legal` π_0)
$(sm\text{-}binprim_i\ \pi_0)$	(`join` π_0)	(`legal` π_0)

Table 2: Calculations for legal stack machine code

stack machine code to any register machine code. We can check the validity of register assumptions with respect to stack machine code by checking that the assumptions are well defined for all lexical addresses that occur in the code.

Definition 13 *Environment assumptions U are* valid *for π if* (`defined` U π) *is true as specified in Table 3.*

A similar restriction occurs with the stack assumptions. The stack machine code manipulates local computations on the top of the stack. In the register machine code, these stack locations will cached in registers. However, if we cache more locations than there are local computations, we will overlap with the continuation or environment representation. This means that our stack assumptions must be restricted so that we never cache more stack locations than local computations. For a given piece of stack machine code, we can calculate the depth of its local computations and use this to check that we aren't caching too many locations.

π	(`depth` π)	(`defined` $U\ \pi$)
$(goto\ \pi_0)$	(`depth` π_0)	(`defined` $U\ \pi_0$)
$(constant\ \epsilon\ \pi_0)$	(`depth` π_0) -1	(`defined` $U\ \pi_0$)
$(fetch\ \iota\ \pi_0)$	(`depth` π_0) -1	$(U \downarrow \iota) \neq error$ and (`defined` $U\ \pi_0$)
$(assign\ \iota\ \pi_0)$	(`depth` π_0)	$(U \downarrow \iota) \neq error$ and (`defined` $U\ \pi_0$)
$(brf\ \pi_0\ \pi_1)$	(`depth` π_0) $+1$	(`defined` $U\ \pi_0$) and (`defined` $U\ \pi_1$)
$(brfng\ \pi_0\ \pi_1)$	(`depth` π_0) $+1$	(`defined` $U\ \pi_0$) and (`defined` $U\ \pi_1$)
$(close\ \nu\ \pi_0\ \pi_1)$	(`depth` π_1) -1	(`defined`$(\langle m_0, \ldots, m_\nu\rangle \cdot U)\ \pi_0$) and (`defined` $U\ \pi_1$)
(rts)	1	*true*
$(catch\ \pi_0\ \pi_1)$	(`depth` π_0) -1	(`defined` $U\ \pi_0$) and (`defined` $U\ \pi_1$)
$(jmp\ \nu)$	$\nu+1$	*true*
$(jsr\ \nu)$	$\nu+1$	*true*
$(test\text{-}arg\ \pi_0)$	(`depth` π_0)	(`defined` $U\ \pi_0$)
$(pop\ \pi_0)$	(`depth` π_0) $+1$	(`defined` $U\ \pi_0$)
$(sm\text{-}unprim_i\ \pi_0)$	(`depth` π_0)	(`defined` $U\ \pi_0$)
$(sm\text{-}binprim_i\ \pi_0)$	(`depth` π_0) $+1$	(`defined` $U\ \pi_0$)

Table 3: Calculations for valid register assumptions

Definition 14 *Stack assumption S is* valid *for π if $\#S \leq$ (`depth` π) as specified in Table 3.*

We are also interested in specifying a class of valid register assumptions that match our register conventions. For the environment assumptions, this means that the registers contained within them are restricted to R_U, and similarly, the stack assumptions are restricted to use registers from R_S.

Definition 15 *Environment assumptions U are* proper *for π if U is valid for π and $R/U \cap R_S = R_S$.*

Definition 16 *Stack assumption S is* proper *for π if S is valid for π and $R/S \cap R_U = R_U$.*

8.5 The back end

We now present the compiler back end. It takes as input stack machine code, the current stack assumptions, and the current environment assumptions and produces register machine code that corresponds to the stack machine code. For the code generator to work properly, the stack machine code must be legal and the register assumptions proper.

The back end is comprised of four valuations:

1. $\mathcal{G}$ - generates the register machine code for the specified stack machine code. It relies heavily upon $\mathcal{G}'$.
2. $\mathcal{G}'$ - generates register machine code for the stack machine code until it reaches a join point in the stack machine code.
3. $\mathcal{P}$ - pops the top of the stack into a register to cache that stack location.
4. $\mathcal{F}$ - frees stack registers by pushing the registers caching the top of the stack onto the stack.

These valuations rely upon a number of auxiliary functions as well:

1. (`get-reg` R) - allocates a register from a set of free registers, R.
2. ($\texttt{mogrify}_S$ $S\,S'$) - generates the code necessary to transform the machine from stack caching S to caching S'.
3. ($\texttt{mogrify}_U$ $U\,U'$) - generates the code necessary to transform the machine from environment caching U to caching U'.
4. (`rtc` U) - generates the code necessary to cache the top lexical contour in the proper registers.
5. (`lookup` $i\,U$) - generates the code necessary to look up the lexical address, i, in the current environment.

8.5.1 The code generator top level

Given stack machine code and register assumptions, $\mathcal{G}$ generates the corresponding register machine code. It generates code for the stack machine leaves by shuffling the registers into the order specified by our register conventions. For non-leaves it uses $\mathcal{G}'$ which generates code until it reaches a join point. $\mathcal{G}$ then finishes the job by recurring on this join point.

$$\begin{aligned}\mathcal{G}[\![(rts)]\!]SU = \texttt{let}^*\ (S',U') &= (\texttt{exit}\ \nu\, U)\\ \gamma &= (\texttt{mogrify}_U\ U\,U')\\ \gamma' &= (\texttt{mogrify}_S\ S\,S')\\ \texttt{in}\ \gamma\,\S\,\gamma'\,\S\langle(rts)\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{G}[\![(jmp\ \nu)]\!]SU = \texttt{let}^*\ (S',U') &= (\texttt{tailcall}\ \nu\, U)\\ \gamma &= (\texttt{mogrify}_U\ U\,U')\\ \gamma' &= (\texttt{mogrify}_S\ S\,S')\\ n &= (U = (C\cdot U'') \to (\nu - \#C), error\\ \texttt{in}\ \gamma\,\S\,\gamma'\,\S\langle(inc\ n\ up),(mov\ up\ sp),(jmp\ u_0)\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{G}[\![(jsr\ \nu)]\!]SU = \texttt{let}^*\ (S',U') &= (\texttt{call}\ \nu\, U)\\ \gamma &= (\texttt{mogrify}_U\ U\,U')\\ \gamma' &= (\texttt{mogrify}_S\ S\,S')\\ \texttt{in}\ \gamma\,\S\,\gamma'\,\S\langle(jmp\ u_0)\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{G}[\![\pi]\!]SU = \texttt{let}^*(\gamma,\pi',S',U') &= \mathcal{G}'[\![\pi]\!]SU\\ \gamma' &= \mathcal{G}[\![\pi']\!]S'U'\\ \texttt{in}\,\gamma\,\S\,\gamma'\end{aligned}$$

8.5.2 The main code generator

$\mathcal{G}'$ does the majority of the code generation. It generates code until it runs into a join point at which time it returns four values:

1. The current sequence of generated code
2. The join point (i.e. the stack machine code left to be translated)
3. The current stack assumptions

4. The current environment assumption

These four values constitute the state necessary to generate the rest of the code for the stack machine program. The reason we must stop generating code at join points is to handle the splits and joins that occur in the program graph.

Recall that a split is introduced by a *brf* instruction and the branches rejoin at a common point identified by a *goto* instruction. To generate the code for the *brf*, we must generate code for both branches followed by the code for the join point. The problem that arises is that the register assumptions at the end of the *true* branch may be completely different from the assumptions at the end of the *false* branch. We must resolve these differences before can generate the code for the join point. Using $\mathcal{G}'$, we can generate the code for each branch and then look at the resultant assumptions before generating the code for the join point. This gives us a great deal of flexibility in resolving the differences in the assumptions. Our approach is to add the code to the end of the *true* branch necessary to transform its assumptions into the assumptions at the end of the *false* branch. We then generate the code for the join point using the assumptions from the *false* branch.

$$\mathcal{G}'[\![(goto\ \pi_0)]\!]SU = (\langle\,\rangle, \pi_0, S, U)$$

$$\begin{aligned}\mathcal{G}'[\![(constant\ \epsilon\,\pi_0)]\!]SU = {} & (R_S/S = \emptyset) \to \mathcal{F}[\![(constant\ \epsilon\,\pi_0)]\!]SU,\\ & \texttt{let}^*\ s = (\texttt{get-reg}\,R_S/S)\\ & \qquad (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]((s, \mathit{false}) :: S)U\\ & \quad \texttt{in}\ ((movi\ \epsilon\, s) :: \gamma, \pi, S', U')\end{aligned}$$

$$\begin{aligned}\mathcal{G}'[\![(fetch\ \iota\,\pi_0)]\!]SU = {} & (R_S/S = \emptyset) \to \mathcal{F}[\![(fetch\ \iota\,\pi_0)]\!]SU,\\ & \texttt{let}^*\ s = (\texttt{get-reg}\,R_S/S)\\ & \qquad (\gamma, U') = (\texttt{lookup}\ \iota\, U)\\ & \qquad (u, \beta) = U' \downarrow \iota\\ & \qquad (\gamma', \pi, S', U'') = \mathcal{G}'[\![\pi_0]\!]((s, \mathit{false}) :: S)U'\\ & \quad \texttt{in}\ (\gamma \,\S \langle (mov\ u\, s)\rangle \,\S\, \gamma', \pi, S', U'')\end{aligned}$$

$$
\begin{aligned}
\mathcal{G}'[\![(assign(d,o)\,\pi_0)]\!]SU = {} & (S \neq s :: S') \rightarrow \mathcal{P}[\![(assign\ \iota\,\pi_0)]\!]SU, \\
& (d \neq 0) \rightarrow \mathtt{let}^*\ (\gamma, U') = (\mathtt{lookup}\,(d-1,0)\,U) \\
& \qquad (u,\beta) = U' \downarrow (d-1,0) \\
& \qquad (\gamma', \pi, S', U'') = \mathcal{G}'[\![\pi_0]\!]SU[m/(d,o)] \\
& \quad \mathtt{in}\ (\gamma\,\S\langle(st\ s\ u(o))\rangle\,\S\,\gamma', \pi, S', U''), \\
& (U \downarrow (0,o) = m) \rightarrow \\
& \quad \mathtt{let}\ (\gamma', \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]SU \\
& \quad \mathtt{in}\ ((st\ s\ \ up(o)) :: \gamma, \pi, S', U'), \\
& \quad \mathtt{let}\ (\gamma', \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]SU[(u_o, false)/(0,o)] \\
& \quad \mathtt{in}\ ((mov\ s\,u_o) :: \gamma, \pi, S', U')
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}'[\![(brf\,\pi_0\,\pi_1)]\!]SU = {} & (S \neq (s,\beta) :: S_0) \rightarrow \mathcal{P}[\![(brf\,\pi_0\,\pi_1)]\!]SU, \\
& \mathtt{let}^*\ (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]S_0U \\
& \qquad (\gamma', \pi, S'', U'') = \mathcal{G}'[\![\pi_1]\!]S_0U \\
& \qquad \gamma'' = \gamma\,\S(\mathtt{mogrify}_S\ S'\,S'')\,\S(\mathtt{mogrify}_U\ U'\,U'') \\
& \quad \mathtt{in}\ ((brf(\#\gamma''+1)\,s) :: \gamma''\,\S\langle(goto\ \#\gamma')\rangle\,\S\,\gamma', \pi, S'', U'')
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}'[\![(brfng\,\pi_0\,\pi_1)]\!]SU = {} & (S \neq (s,\beta) :: S_0) \rightarrow \mathcal{P}[\![(brfng\,\pi_0\,\pi_1)]\!]SU, \\
& \mathtt{let}^*\ \gamma = \mathcal{G}[\![\pi_0]\!]S_0U \\
& \qquad (\gamma', \pi, S'', U'') = \mathcal{G}'[\![\pi_1]\!]S_0U \\
& \quad \mathtt{in}\ ((brfng\ \#\gamma\ s) :: \gamma\,\S\,\gamma', \pi, S'', U'')
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{G}'[\![(close\ \nu\,\pi_0\,\pi_1)]\!]SU = {} & (R_S/S = \emptyset) \rightarrow \mathcal{F}[\![(close\ \nu\,\pi_0\,\pi_1)]\!]SU, \\
& \mathtt{let}^*\ s = (\mathtt{get\text{-}reg}\,R_S/S) \\
& \qquad (S', U') = (\mathtt{entry}\ \nu\,U) \\
& \qquad \gamma = \mathcal{G}[\![\pi_0]\!]S'U' \\
& \qquad (\gamma', \pi_1, S'', U'') = \mathcal{G}'[\![\pi]\!]((s, false) :: S)U \\
& \quad \mathtt{in}\ ((close\ \#\gamma\ s\,\nu) :: \gamma\,\S\,\gamma', \pi, S'', U'')
\end{aligned}
$$

$$
\mathcal{G}'[\![(rts)]\!]SU = (\langle\,\rangle, (rts), S, U)
$$

$$\begin{aligned}\mathcal{G}'[\![(\mathit{catch}\ \pi_0\ \pi_1)]\!]SU = \quad & \texttt{let*}\ \gamma = (\texttt{mogrify}_S\ S\,\langle\,\rangle)\\ & \qquad \gamma' = \mathcal{G}[\![\pi_1]\!]\langle\,\rangle U\\ & \qquad (S', U') = (\texttt{return}\ U')\\ & \qquad (\gamma'', U') = (\texttt{rtc}\ U')\\ & \qquad (\gamma''', \pi, S'', U'') = \mathcal{G}'[\![\pi_0]\!]S'U'\\ & \quad \texttt{in}\ (\gamma\,\S\langle(\mathit{close}\ \#\gamma'\ s_0\ 1)\rangle\,\S\,\gamma'\,\S\,\gamma''\,\S\,\gamma''', \pi, S'', U'')\end{aligned}$$

$$\mathcal{G}'[\![(\mathit{jmp}\ \nu)]\!]SU = (\langle\,\rangle, (\mathit{jmp}\ \nu), S, U)$$

$$\mathcal{G}'[\![(\mathit{jsr}\ \nu)]\!]SU = (\langle\,\rangle, (\mathit{jsr}\ \nu), S, U)$$

$$\mathcal{G}'[\![(\mathit{test\text{-}arg}\ \pi_0)]\!]SU = \mathcal{G}'[\![\pi_0]\!]SU$$

$$\mathcal{G}'[\![(\mathit{pop}\ \pi_0)]\!]SU = (S \neq (s, \beta) :: S_0) \rightarrow \mathcal{P}[\![(\mathit{pop}\ \pi_0)]\!]SU, \mathcal{G}'[\![\pi_0]\!]S_0U$$

$$\begin{aligned}\mathcal{G}'[\![(\mathit{sm\text{-}unprim}_i\ \pi_0)]\!]SU = (S \neq (s, \beta) :: S_0) \rightarrow\ & \mathcal{P}[\![(\mathit{sm\text{-}unprim}_i\ \pi_0)]\!]SU,\\ & \texttt{let}\ (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]((s, \mathit{false}) :: S_0)U\\ & \quad \texttt{in}\ ((\mathit{rm\text{-}unprim}_i\ s) :: \gamma, \pi, S', U')\end{aligned}$$

$$\begin{aligned}\mathcal{G}'[\![(\mathit{sm\text{-}binprim}_i\ \pi_0)]\!]SU = (S \neq (s_1, \beta) :: (s_0, \beta) :: S_0) \rightarrow\ & \mathcal{P}[\![(\mathit{sm\text{-}binprim}_i\ \pi_0)]\!]SU,\\ & \texttt{let}\ (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi_0]\!]((s_1, \mathit{false}) :: S_0)U\\ & \quad \texttt{in}\ ((\mathit{rm\text{-}binprim}_i\ s_0\ s_1) :: \gamma, \pi, S', U')\end{aligned}$$

8.5.3 Caching local computations

This is the inverse of $\mathcal{F}$. It pops the top value of the stack into a register and adds the information to the stack assumptions. If there is not free stack register, an error is signaled[2].

$$\mathcal{P}[\![\pi]\!]SU = (R_S/S = \emptyset) \rightarrow \mathit{error}$$

[2] All uses of $\mathcal{P}$ are guarded to ensure that no error is signaled.

$$\begin{array}{l} \texttt{let*}\ s = (\texttt{get-reg}\, R_S/S) \\ \quad (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi]\!](S \,\S\, \langle(s, \mathit{false})\rangle)U \\ \ \texttt{in}\ ((\mathit{pop}\ s) :: \gamma, \pi, S', U') \end{array}$$

8.5.4 Spilling local computations

This valuation is used by $\mathcal{G}'$ to free a register. It does so by taking the very last register used to cache the top of stack and pushing its value on the top of the stack. If no registers are being used to cache the top of stack, an error is signaled[3].

$$\begin{array}{l} \mathcal{F}[\![\pi]\!]SU = (S \neq S_0 \,\S\, \langle(s, \beta)\rangle) \rightarrow \mathit{error} \\ \qquad\qquad \texttt{let}\ (\gamma, \pi, S', U') = \mathcal{G}'[\![\pi]\!]S_0U \\ \qquad\qquad\ \ \texttt{in}\ ((\mathit{push}\ s) :: \gamma, \pi, S', U') \end{array}$$

8.6 Allocator auxiliaries

8.6.1 Allocating a free register

Given a set of free registers, this randomly chooses one. If there are no free registers, it signals an error[4].

$$(\texttt{get-reg}\ R) = (R \neq (\{r\} \cup R')) \rightarrow \mathit{error}, r$$

8.6.2 Transforming the top of the stack

The auxiliary function $(\texttt{mogrify}_S\ S\,S')$ produces a code sequence which translates the top of the stack conforming to S into one conforming to S'. How it does this is dependent upon the relative sizes of S and S':

- If $S = S'$ it need only shuffle the registers to transform S into S'.

[3] All uses of $\mathcal{F}$ are guarded to ensure that no error is signaled.
[4] The uses of `get-reg` are guarded to ensure that no error is signaled.

- If $S > S$ it must push registers onto the stack until their sizes are equal and then proceed as before.
- If $S < S'$ we must pop the top of the stack into more registers until their sizes are equal and, again, proceed as before.

$\texttt{mogrify}_S$ does this by dispatching to one of three help functions. Each help function handles one of the cases we just described.

$$\begin{aligned}(\texttt{mogrify}_S\ S\,S') = {} & (\#S = \#S') \rightarrow (\texttt{mogrify}_{S=}\ S\,S'),\\ & (\#S < \#S') \rightarrow (\texttt{mogrify}_{S<}\ S\,S'), (\texttt{mogrify}_{S>}\ S\,S')\end{aligned}$$

Transmogrify$_=$

$\texttt{mogrify}_{S=}$ is used to transform the the top of stack when the original stack assumption has the same number of locations cached as the new one.

$$\begin{aligned}(\texttt{mogrify}_{S=}\ S\,S') = {} & (S = \langle\,\rangle) \rightarrow \langle\,\rangle,\\ & \texttt{let}^*\ (s,\beta) = S \downarrow 1\\ & \qquad (s',\beta') = S' \downarrow 1\\ & \qquad \gamma = (\texttt{mogrify}_{S=}\,(S \dagger 1)\,(S' \dagger 1))\\ & \quad \texttt{in}\ (s = s') \rightarrow \gamma,\\ & \qquad (s \in R_S/S) \rightarrow (mov\ s\ s') :: \gamma, (sw\ s\ s') :: \gamma\end{aligned}$$

Transmogrify$_>$

$\texttt{mogrify}_{S>}$ is used to transform the the top of stack when the original stack assumption has more locations cached then the new one.

$$(\texttt{mogrify}_{S>}\ S\,S') = (S \neq S_0 :: (s, \beta)) \rightarrow error,$$
$$(push\, s) :: (\texttt{mogrify}_S\ S_0\, S')$$

Transmogrify$_<$

$\texttt{mogrify}_{S<}$ is used to transform the the top of stack when the new stack assumption has more locations cached then the original one.

$$(\texttt{mogrify}_{S<}\ S\,S') = \texttt{let}\ \ s = (\texttt{get-reg}\ R_S/S)$$
$$\texttt{in}\ (pop\, s) :: (\texttt{mogrify}_S\, (S \,\S\langle(s, \mathit{false})\rangle)\, S')$$

8.6.3 Caching the top lexical contour

`rtc` returns the code necessary to load the top lexical contour as well as the resultant register assumptions.

$$(\texttt{rtc}\ U) = (U \neq (C \cdot U')) \rightarrow (\langle\,\rangle, U), ((\texttt{rtc}_l\ 0\,\#C), (\langle(u_0, true), \ldots, (u_{\#C}, true)\rangle \cdot U'))$$

$$(\texttt{rtc}_l\ i\, n) = (i = n) \rightarrow \langle(ld\ up(i)\, u_i)\rangle,$$
$$(i < n) \rightarrow (ld\ up(i)\, \mu_i) :: (\texttt{rtc}_l\, (i+1)\, n), error$$

8.6.4 Looking up lexical addresses

`lookup` returns two values. The first value is the code to move the lexical addresses into a register and the second is the environment assumptions that occur after the code has been executed.

It works by first checking if the lexical address is already cached, in which case its done. Otherwise it looks up the static chain pointer from the previous contour and the uses that to load the lexical address. This code relies upon the fact that the top contour is cached in registers (in particular, that the topmost static chain pointer is in u_0.).

$$
\begin{aligned}
(\texttt{lookup}\,(d,o)\,U) = (U \downarrow (d,o) = (u,\beta)) \rightarrow (\langle\,\rangle, U),&\\
\texttt{let}^*\ (\gamma, U') &= (\texttt{lookup}\,(d-1,0)\,U)\\
(u,\beta) &= U' \downarrow (d-1,0)\\
(u',\beta') &= (R_U/U' = \emptyset) \rightarrow u, (\texttt{get-reg}\ R_U/U')\\
\texttt{in}\ (u \neq u') \rightarrow (\gamma ::(ld\,u(o)\,u'), U[(u', true)/(d,o)]),&\\
(\gamma ::(deref\,u(o)), U[m/(d-1,0)][(u, true)/(d,o)])&
\end{aligned}
$$

8.6.5 Transforming the environment

$\texttt{mogrify}_U$ returns the code necessary to go from environment register caching U to caching U'. It does this in three phases:

- It begins by storing all of the variables that are not clean in U but are clean in U'.
- Next it loads all of the variable that are in registers in U' that are not in registers in U.
- Finally, it swaps the registers around so that U and U' match.

$\texttt{mogrify}_U$ depends upon three help functions that implement each of the cases described above.

$$
\begin{aligned}
(\texttt{mogrify}_U\ U\,U') = \texttt{let}^*\ (\gamma, U) &= (\texttt{store-em}\ U\,U')\\
(\gamma', U) &= (\texttt{load-em}\ U\,U')\\
(\gamma'', U) &= (\texttt{swap-em}\ U\,U')\\
\texttt{in}\ \gamma\,\S\,\gamma'\,\S\,\gamma''&
\end{aligned}
$$

`store-em`

`store-em` stores all of the variables that are not clean in U but are clean in U'. It does so by calling $\texttt{store-em}_U$ which iterates over the contours of U. For each contour, $\texttt{store-em}_U$ calls $\texttt{store-em}_S$ which will generate the code to store the variables for the contour.

$$\begin{aligned}
(\texttt{store-em}\; U\, U') = \texttt{let}^*\; (\gamma, U) &= (\texttt{store-em}_U\; 0\, U\, U') \\
&\texttt{in}\; (\gamma = \langle\,\rangle) \rightarrow (\gamma, U), ((\mathit{mov}\; \mathit{up}\; u_0) :: \gamma \,\S\, \langle(\mathit{ld}\; \mathit{up}(0)\, u_0)\rangle, U)
\end{aligned}$$

$$\begin{aligned}
(\texttt{store-em}_U\; i\, U\, U') = (i = \#U) &\rightarrow (\langle\,\rangle, \langle\,\rangle), \\
&\texttt{let}^*\; (\gamma, S) = (\texttt{store-em}_S\; 0\, (U \downarrow i)\, (U' \downarrow i)) \\
&\qquad (\gamma', U) = (\texttt{store-em}_U\, (i+1)\, U\, U') \\
&\quad \texttt{in}\; (\gamma' = \langle\,\rangle) \rightarrow (\gamma, S \cdot U), (\gamma \,\S\, \langle(\mathit{deref}\; u_0)\rangle \,\S\, \gamma', S \cdot U)
\end{aligned}$$

$$\begin{aligned}
(\texttt{store-em}_S\; i\, S\, S') = (i = \#S) &\rightarrow (\langle\,\rangle, \langle\,\rangle), \\
&\texttt{let}^*\; a = (S \downarrow i)) \\
&\qquad a' = (S' \downarrow i)) \\
&\quad \texttt{in}\; (a \neq (u, \beta)) \rightarrow (\gamma, a :: S), \\
&\qquad \beta \rightarrow ((a' = m) \rightarrow (\gamma, a' :: S), (\gamma, a :: S)), \\
&\qquad\quad (a' \neq (u', \beta')) \rightarrow ((\mathit{st}\; u\; u_0(i)) :: \gamma, a' :: S), \\
&\qquad\qquad \beta \rightarrow ((\mathit{st}\; u\; u_0(i)) :: \gamma, (u, \mathit{true}) :: S), \\
&\qquad\qquad\quad (\gamma, a :: S)
\end{aligned}$$

`load-em`

`load-em` loads all of the variable that are in registers in U' that are not in registers in U. It uses an approach similar to `store-em` by calling $\texttt{load-em}_U$ which iterates over the contours of U. For each contour, $\texttt{load-em}_U$ calls $\texttt{load-em}_S$ which will generate the code to load the variables for the contour.

$$
\begin{aligned}
&(\texttt{load-em}\ U\,U') = \texttt{let}^*\ (\gamma, U) = (\texttt{load-em}_U\ 0\,(R_U/U)U\,U')\\
&\qquad\qquad\qquad\qquad \texttt{in}\ (\gamma = \langle\,\rangle) \rightarrow (\gamma, U), ((\mathit{mov\ up}\ u_0) :: \gamma\,\S\langle(\mathit{ld\ up}(0)\,u_0)\rangle, U)
\end{aligned}
$$

$$
\begin{aligned}
(\texttt{load-em}_U\ i\,f\,U\,U') = (i = \#U) &\rightarrow (\langle\,\rangle, \langle\,\rangle),\\
&\texttt{let}^*\ (f, \gamma, S) = (\texttt{load-em}_S\ 0\,f\,(U \downarrow i)\,(U' \downarrow i))\\
&\qquad (\gamma', U) = (\texttt{load-em}_U\,(i+1)\,f\,U\,U')\\
&\quad \texttt{in}\ (\gamma' = \langle\,\rangle) \rightarrow (\gamma, S \cdot U), (\gamma\,\S\langle(\mathit{deref}\,u_0)\rangle\,\S\,\gamma', S \cdot U)
\end{aligned}
$$

$$
\begin{aligned}
&(\texttt{load-em}_S\ i\,f\,S\,S') =\\
&\quad (i = \#S) \rightarrow (f, \langle\,\rangle, \langle\,\rangle),\\
&\qquad \texttt{let}^*\ a = (S \downarrow i))\\
&\qquad\qquad a' = (S' \downarrow i))\\
&\qquad\quad \texttt{in}\ (a = m) \rightarrow (a' \neq m) \rightarrow \texttt{let}^*\ u = (\texttt{get-reg}\ f)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad f = f - \{u\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (f, \gamma, S) = (\texttt{load-em}_S\,(i+1)\,f\,S\,S')\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in}\ (f, (\mathit{ld}\ u_0(i)\ u) :: \gamma, (u, \mathit{true}) :: S),\\
&\qquad\qquad\qquad\qquad \texttt{let}^*\ (f, \gamma, S) = (\texttt{load-em}_S\,(i+1)\,f\,S\,S')\\
&\qquad\qquad\qquad\qquad\quad \texttt{in}\ (f, \gamma, a :: S),\\
&\qquad\qquad \texttt{let}^*\ (f, \gamma, S) = (\texttt{load-em}_S\,(i+1)\,f\,S\,S')\\
&\qquad\qquad\quad \texttt{in}\ (f, \gamma, a :: S)
\end{aligned}
$$

`swap-em`

`swap-em` swaps the registers around so that U and U' match. Again, it uses an approach similar to `store-em` and `load-em` by calling $\texttt{swap-em}_U$ which iterates over the contours of U. For each contour, $\texttt{swap-em}_U$ calls $\texttt{swap-em}_S$ which will generate the code to swap the registers caching the variables for the contour.

$$(\texttt{swap-em}\ U\ U') = (\texttt{swap-em}_U\ 0\ U\ U')$$

$$\begin{aligned}(\texttt{swap-em}_U\ i\ U\ U') = {} & (i = \#U) \to \langle\,\rangle, \\ & (\texttt{swap-em}_S\ 0\ U'\ (U \downarrow i)\ (U' \downarrow i))\,\S(\texttt{swap-em}_U\ (i+1) U\ U')\end{aligned}$$

$$\begin{aligned}
&(\texttt{swap-em}_S\ i\ U'\ S\ S') = \\
&\quad (i = \#S) \to \langle\,\rangle, \\
&\qquad \texttt{let}^*\ a = (S \downarrow i)) \\
&\qquad\qquad a' = (S' \downarrow i)) \\
&\qquad\qquad \gamma = (\texttt{swap-em}_S\ (i+1)\ U'\ S\ S') \\
&\qquad\quad \texttt{in}\ (a \neq (u, \beta)) \to \gamma, \\
&\qquad\qquad\quad (a' \neq (u', \beta')) \to \gamma, \\
&\qquad\qquad\qquad (u \notin (R_U/U')) \to (sw\ u\ u') :: \gamma, (mov\ u\ u') :: \gamma
\end{aligned}$$

8.7 Correctness of the back end

We will now show the correctness of the register allocator. The top-level correctness criterion is fairly simple: For any stack-machine program π, we should have that $(S, U) \models \mathcal{G}[\![\pi]\!]SU \simeq_P \pi$. However, we need to state induction hypotheses for all valuations of the allocator. Furthermore, we must deal with complications of branching and joining code. The allocator can only be expected to work properly on code in which *goto* instructions are properly matched. We begin by introducing some basic properties of the auxiliaries used for manipulating the register caching.

Lemma 10 *If S and U are proper for π, $(S, U) \models \gamma \simeq_P \pi$, and S' is valid for π then $(S', U) \models (\texttt{mogrify}_S\ S'\ S)\,\S\,\gamma \simeq_P \pi$.*

Proof: Structural induction on S and S'. $\square$

Lemma 11 *If S and U are proper for π, $U = (\langle a_1, \ldots, a_n\rangle \cdot U')$, $\forall\ (0 \leq i \leq n)\ (a_i \neq (u, \mathit{false})$ and $u_i \notin U)$, and $(S, (\langle (u_0, \mathit{true}), \ldots, (u_n, \mathit{true})\rangle \cdot U')) \models \gamma \simeq_P \pi$ then $(S, U) \models (\texttt{rtc}\ U) :: \gamma \simeq_P \pi$*

Proof: Induction on U. □

Lemma 12 *If S and U are proper for π, then $(\texttt{lookup}\ \iota\, U) = (\gamma, U')$ s.t.:*

1. *U' is proper for π*
2. *$U' \approx U$*
3. *$U' \downarrow \iota = (u, \mathit{true})$*
4. *for any γ', if $(S, U') \models \gamma' \simeq_P \pi$ then $(S, U) \models \gamma \,\S\, \gamma' \simeq_P \pi$*

Proof: Induction on ι. □

Lemma 13 *If $(S, U) \models \gamma \simeq_P \pi$, $U' \approx U$ and $(R/S) \cap (R/U') = \emptyset$ then $(S, U') \models (\texttt{mogrify}_U\ U'\, U) \,\S\, \gamma \simeq_P \pi$.*

Proof: Structural induction on the common shape of U and U'. □

Now we can finally state and prove the correctness of the back end.

Theorem 5 *If S and U are proper for π and π is legal then*

- *$\mathcal{G}[\![\pi]\!]SU = \gamma$ s.t. $(S, U) \models \gamma \simeq_P \pi$*
- *$\mathcal{G}'[\![\pi]\!]SU = (\gamma, \pi', S', U')$ s.t.*
  1. *π' is the join point of π*
  2. *S' and U' are proper for π'*
  3. *$U' \approx U$*

4. *for any* γ', *if* $(S', U') \models \gamma' \simeq_P \pi'$ *then* $(S, U) \models \gamma \,\S\, \gamma' \simeq_P \pi$

- $\mathcal{P}[\![\pi]\!]SU = (\gamma, \pi', S', U')$ *s.t.*

  1. π' *is the join point of* π
  2. S' *and* U' *are proper for* π'
  3. $U' \approx U$
  4. *for any* γ', *if* $(S', U') \models \gamma' \simeq_P \pi'$ *then* $(S, U) \models \gamma \,\S\, \gamma' \simeq_P \pi$

- $\mathcal{F}[\![\pi]\!]SU = (\gamma, \pi', S', U')$ *s.t.*

  1. π' *is the join point of* π
  2. S' *and* U' *are proper for* π'
  3. $U' \approx U$
  4. *for any* γ', *if* $(S', U') \models \gamma' \simeq_P \pi'$ *then* $(S, U) \models \gamma \,\S\, \gamma' \simeq_P \pi$

Proof: By analysis of each instruction. Here we show *brf*, the trickiest of the stack machine instructions to handle.

Assume we have that S and U are proper for $(\mathit{brf}\,\pi_0\,\pi_1)$ and that $(\mathit{brf}\,\pi_0\,\pi_1)$ is legal. We must show that $\mathcal{G}'[\![(\mathit{brf}\,\pi_0\,\pi_1)]\!]SU = (\gamma, \pi', S', U')$ s.t.

1. π' is the join point of $(\mathit{brf}\,\pi_0\,\pi_1)$
2. S' and U' are proper for π'
3. $U' \approx U$
4. for any γ', if $(S', U') \models \gamma' \simeq_P \pi'$ then $(S, U) \models \gamma \,\S\, \gamma' \simeq_P \pi$

There are two cases to consider.

- $S \neq (s, \beta) :: S'$
  - $\mathcal{G}'[\![(\mathit{brf}\,\pi_0\,\pi_1)]\!]SU = \mathcal{P}[\![(\mathit{brf}\,\pi_0\,\pi_1)]\!]SU$

  - Hence the result is established by induction.

- $S = (s, \beta) :: S'$

  - $(brf\,\pi_0\,\pi_1)$ is legal thus $(\texttt{join}(brf\,\pi_0\,\pi_1)) = (\texttt{join}\ \pi_0) = (join\,\pi_1) = \pi'$.
  - S and U are proper for $(brf\,\pi_0\,\pi_1)$ hence S' and U are proper for π_0. $(brf\,\pi_0\,\pi_1)$ is legal hence π_0 is legal. By induction $\mathcal{G}'[\![\pi_0]\!]S'U = (\gamma_0, \pi', S_0, U_0)$ s.t.

    1. π' is the join point of π_0.
    2. S_0 and U_0 are proper for π'.
    3. $U_0 \approx U$
    4. for any γ', if $(S_0, U_0) \models \gamma' \simeq_P \pi'$ then $(S', U) \models \gamma_0 \S\, \gamma' \simeq_P \pi$

  - S and U are proper for $(brf\,\pi_0\,\pi_1)$ hence S' and U are proper for π_1. $(brf\,\pi_0\,\pi_1)$ is legal hence π_1 is legal. By induction $\mathcal{G}'[\![\pi_1]\!]S'U = (\gamma_1, \pi', S_1, U_1)$ s.t.

    1. π' is the join point of π_0.
    2. S_1 and U_1 are proper for π'.
    3. $U_1 \approx U$
    4. for any γ', if $(S_1, U_1) \models \gamma' \simeq_P \pi'$ then $(S', U) \models \gamma_1 \S\, \gamma' \simeq_P \pi$

  - By Lemmas 10 and 13 if $(S_1, U_1) \models \gamma' \simeq_P \pi'$ then for $\gamma_2 = \gamma_0 \S (\texttt{mogrify}_S\ S_0\, S_1) \S (\texttt{mogrify}_U\, U_0\, U_1)$ we can conclude that $(S_0, U_0) \models \gamma_2 \S\, \gamma' \simeq_P \pi'$
  - Hence

    1. π' is the join point of $(brf\,\pi_0\,\pi_1)$.
    2. S_1 and U_1 are proper for π'.
    3. $U_1 \approx U$
    4. for any γ', if $(S_1, U_1) \models \gamma' \simeq_P \pi'$ then $(S, U) \models (brf\ \#\gamma_2{+}1) :: \gamma_2 \S \langle (goto\ \#\gamma_1) \rangle \S\, \gamma_1 \S\, \gamma' \simeq_P (brf\,\pi_0\,\pi_1)$.

    and the result is established.

□

8.8 Factorial example

If Figure 15 we recall the stack machine code for the factorial example from Chapter 5. In Figures 16 and 17 we show the register machine code generated from the stack machine code. We have indented the generated code and included comments to relate it to the original stack machine code.

Note that in the outermost call to $fact$, we must save the environment frame so that we can refer to $fact$ again. For the inner call to fact, the environment does not need to be save and only a bit of register shuffling is necessary to set up the call.

```
;;; Push the argument to the outer closure
(constant 0
;;; Push outer closure
(close 1
  ;; Push factorial function
  (close 2
    ; Check value of i
    (fetch (0 . 2)
    (primitive zero? 1
    (brfng
      ; If i is zero return a
      (fetch (0 . 1)
      (rts))
    ; If i is not 0
    ; Push i - 1
    (fetch (0 . 2)
    (constant 1
    (primitive - 2
    ; Push i * a
    (fetch (0 . 1)
    (fetch (0 . 2)
    (primitive * 2
    ; Get factorial and jump to it.
    (fetch (1 . 1)
     (jmp 2)))))))))))
  ;; Assign factorial to local variable
  (assign (0 . 1)
  ;; Clear result of assign
  (pop
  ;; Push argument i to factorial
  (constant 4
  ;; Push argument a to factorial
  (constant 1
  ;; Get factorial and jump to it
  (fetch (0 . 1)
  (jsr 2)))))))
;;; Jump to outer closure
(jsr 1)))
```

Figure 15: Compiled Iterative Factorial Program

```
;;; Move 0 into stack register s0
((movi 0 s0)
 ;;; Move outer closure into stack register s1
 (close 28 s1 1)
   ;; Move factorial function to stack register s0
   (close 17 s0 2)
     ; Check value of i
     (mov u2 s0)
     (prim zero? s0)
     (brfng 2 s0)
       ; If i is zero return a
       (mov u1 s0)
       (rts)
     ; If i is not 0
     ; Move  i - 1 to stack register s1
     (mov u2 s0)
     (movi 1 s1)
     (prim - s0 s1)
     ; Move  i * 1 to stack register s2
     (mov u1 s0)
     (mov u2 s2)
     (prim * s0 s2)
     ; Move factorial into register s0
     (ld (u0 1) u3)
     (mov u3 s0)
     (mov s0 u0)
     ; Move stack registers into environment registers
     (mov s2 u1)
     (mov s1 u2)
     ; Jump to factorial
     (jmp u0)
   ;; Assign factorial to local variable
   (mv s0 u1)
   ;; Move argument i to stack register s0
   (movi 4 s0)
```

Figure 16: Compiled Iterative Factorial Program

```
  ;; Move argument a to stack register s1
  (movi 1 s1)
  ;; Move factorial to stack register s0
  (mov u1 s2)
  ;; Save current environment frame
  (st u0 (up 0))
  (st u1 (up 1))
  ;; Move stack registers into environment registers
  (mov s2 u0)
  (mov s1 u1)
  (mov s0 u2)
  ;; Jump to factorial
  (jmp u0)
;;; Move stack registers into environment registers
(mov s1 u0)
(mov s0 u1)
;;; Jump to outer closure
(jmp u0))
```

Figure 17: Compiled Iterative Factorial Program (continued)

Chapter 9

Conclusions

Our goal in this dissertation was to present a methodology for language implementation that mirrored traditional methods yet allowed for verification of the implementation. Our methodology advocated the use of denotational semantics to verify compiler front ends and operational semantics to verify compiler back ends. In particular, to write and verify an implementation of programming language given a traditional denotational definition, we advocate the following approach:

- Derive a compiler front end using the Wand-Clinger approach (chapter 5).
- Switch to operational reasoning by giving an equivalent operational definition of the resultant abstract machine (chapter 5).
- Consider the representation of abstract data in the target machine by use of a hybrid machine and storage-layout relations (chapter 6).
- Relate the code of the abstract machine to the code of the target machine using register assumptions and storage layout relations (chapters 7 and 8).
- Repeat the last step as necessary.

On an intuitive level, our approach seems sound. Denotational semantics assign meaning to programming languages on an abstract level through the use of functions. This complements the traditional view of compiler front ends as the machine independent component of compilers. Similarly, operational semantics assigns language meaning based upon the execution sequence of a particular machine. Its use complements the traditional view of compiler back ends as the machine dependent component of compilers.

9.1 Results

Appealing as our approach may be on an intuitive level, the real issue is how well it performs in practice. We have applied our methodology to three languages:

1. A small language with recursive functions and continuations (in [79])
2. Pure PreScheme (in [46])
3. ν-LISP (in this dissertation)

Our goal was to develop a methodology to verify efficient language implementations without undue formal complexity, so we consider performance on two levels:

1. Methodology efficiency
2. Implementation efficiency

9.1.1 Methodology efficiency

We have applied our methodology to three languages. Each of these languages have had their own distinct features which our techniques have readily adapted to. This gives us confidence that our methodology is widely applicable but more research is indicated.

During the development of ν-LISP compiler, our formalisms on one level of the implementation gave us coherent feedback on how to modify the other levels for an improved compiler[1]. We feel that this feedback is a distinctive feature of our methodology that helps overcome the inherent ad-hoc manner in which compiler back ends are often written. As an example, we changed the representation of continuations when it became apparent that an alternate representation would be considerably more efficient on the register machine. The new representation was easily incorporated into the existing representation structure and only required the revision of one clause of the representation proof.

9.1.2 Implementation efficiency

The ν-LISP compiler described in the dissertation has not been implemented yet so it is difficult to measure its efficiency. However, the Pure PreScheme compiler was implemented so we rely upon [48] to assess the efficiency of implementations based upon our approach:

> Use of a verified compiler need not entail catastrophic performance penalties. On a realistic example, we showed that a verified program compiled by our verified compiler used only about four times more CPU time compared with an unverified version compiled by the optimizing gcc.

9.2 Future work

9.2.1 Global analysis

Incorporating global analysis into the methodology would allow us to perform many of the optimizations utilized by contemporary optimizing compilers. This is a necessary step to make verified implementations competitive with traditional ones and needs to be researched.

9.2.2 Application to other languages

The results indicate that our methodology is flexible enough to extend to other classes of languages (e.g. higher order languages, functional languages, and reflective languages) but this needs to be explored further.

[1] We had similar experiences with the Pure PreScheme compiler as well.

9.2.3 Automation

There are two broad areas that lend themselves to automation in our methodology. The first area is the language implementation process. It should be possible to develop tool support for the methodology based on formalisms we've presented. The second area for automation is the verification process. The proofs that arise from the methodology seem amenable to mechanical proof-checking. We feel that automating the verification process would enhance the validity of the verification and make the methodology appealing to a larger segment of the community.

Bibliography

[1] Abelson, H., Sussman, G.J., and Sussman, J. *Structure and Interpretation of Programs*, MIT Press, Cambridge, MA, 1985.

[2] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] Appel, A.W. "Semantics-Directed Code Generation," *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages* (1985), 315–324.

[4] Appel, A.W. "Runtime Tags Aren't Necessary," *Lisp and Symbolic Computation 2* (1989), 153–162.

[5] Appel, A.W. "A Runtime System," *Lisp and Symbolic Computation 3* (1990), 343–380.

[6] Bahkle, R., and Snelting, G. "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 547–576.

[7] Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[8] Bodwin, J., Bradley, L., Kanda, K., Litle, D., and Pleban, U. "Experience with an Experimental Compiler Generator," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction* (June 1982), 216–229.

[9] Burstall, R.M. "Proving Properties of Programs by Structural Induction," *Computer Journal 12*, 1 (February 1969), 41–48.

[10] Chambers, C. "The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages," Ph.D. Thesis, Stanford University, 1992.

[11] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D., and Norman, A.C. "SKIM — The S, K, I Reduction Machine," *Conference Record of the 1980 ACM Symposium on Lisp and Functional Programming* (August 1980), 128–135.

[12] Clinger, W. "The Scheme 311 Compiler: An Exercise in Denotational Semantics," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (August 1984), 356–364.

[13] Clinger, W., and Rees, J., eds. "Revised^4 Report on the Algorithmic Language Scheme," Indiana University Computer Science Department Technical Report 341, November 1991.

[14] Dybvig, R.K., Hieb, R., and Butler, T "Destination-Driven Code Generation," Indiana University Computer Science Department Technical Report 302, February 1990.

[15] Farmer, W.M., Guttman, J.D., Monk, L.G., Ramsdell, J.D., and Swarup, V. "VLISP: A Verified Language Implementation: FY 91 Year-End Report," The MITRE Corporation Technical Report MTR11232, October 1991.

[16] Fraser, C.W. "Automatic Generation of Code Generators," Ph.D. Thesis, Yale University, 1977.

[17] Friedman, D.P., Wand, M., and Haynes, C.T. *Essentials of Programming Languages*, MIT Press, Cambridge, MA, and McGraw-Hill, Chicago, 1992.

[18] Ganapathi, M., and Fischer, C.N. "Description-Driven Code Generation Using Attribute Grammars," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages* (1982), 108–119.

[19] Glanville, R.S., and Graham, S.L. "A New Method for Compiler Code Generation," *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages* (1978), 231–240.

[20] Goldberg, B. "Tag-Free Garbage Collection for Strongly Typed Programming Languages," *Proceedings of the ACM SIGPLAN '91 Symposium on Programming Language Design and Implementation* (June 1991), 165–176.

[21] Gunter, C.A. *Semantics of Programming Languages — Structures and Techniques*, MIT Press, Cambridge, MA, 1992.

[22] Hannan, J. "Making Abstract Machines Less Abstract," *Functional Programming Languages and Computer Architecture, 5th ACM Conference* (J. Hughes, ed.) *Springer Lecture Notes in Computer Science 523* (1991), 618–635.

[23] Hoare, C.A.R. "Proving Correctness of Data Representations," *ACTA Informatica 1* 1972, 271–281.

[24] Hoffman, C.M. "Design and Correctness of a Compiler for a Non-Procedural Language," *ACTA Informatica 9* 1978, 217–241.

[25] Hughes, R.J.M. "Super-Combinators: A New Implementation Method for Applicative Languages," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming* (August 1982), 1–10.

[26] Johnson, S.C. "Yacc: Yet Another Compiler-Compiler," Bell Laboratories Technical Report CSTR32, 1975.

[27] Jones, N.D., and Muchnick, S. "A Fixed-Program Machine for Combinator Expression Evaluation," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming* (August 1982), 11–20.

[28] Kelsey, R. "Compilation By Program Transformation," Ph.D. Thesis, Yale University, 1989.

[29] Kelsey, R. "PreScheme: A Dialect of Scheme for Systems Programming," Northeastern University Technical Report, in preparation.

[30] Koopman, P.J., and Lee, P. "A Fresh Look at Combinator Graph Reduction," *Proceedings of the ACM SIGPLAN '89 Symposium on Programming Language Design and Implementation* (1989), 110–119.

[31] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. "Orbit: An Optimizing Compiler for Scheme," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction* (June 1986), 219–233.

[32] Kranz, D. "Orbit: An Optimizing Compiler for Scheme," Ph.D Thesis, Yale University, 1988.

[33] Landin, P.J. "The Mechanical Evaluation of Expressions," *Computer Journal 6*, (1964), 308–320.

[34] Lee, P. "The Automatic Generation of Realistic Compilers from High-Level Semantics Descriptions," Ph.D. Thesis, University of Michigan, 1986.

[35] Lee, P., and Pleban, U. "On the Use of LISP in Implementing Denotational Semantics," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming* (June 1986), 233–248.

[36] Lee, P., and Pleban, U. "A Realistic Compiler Generator Based on High-Level Semantics," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (1987), 284–295.

[37] Lee, P., and Pleban, U. "An Automatically Generated, Realistic Compiler for an Imperative Programming Language," *Proceedings of the ACM SIGPLAN '88 Symposium on Programming Language Design and Implementation* (1988), 222–227.

[38] Lee, P. *Topics in Advanced Language Implementation*, MIT Press, Cambridge, MA, 1991.

[39] McCarthy, J., and Painter, J. "Correctness of a Compiler for Arithmetic Expressions," in *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science* (J. T. Schwartz, ed.) American Mathematical Society, Providence, RI, 1967, 33–41.

[40] Milne, R., and Strachey C. *A Theory of Programming Language Semantics*, Chapman and Hall, London, and Wiley, New York, 1976.

[41] Mosses, P. "SIS — Semantics Implementation System: Reference Manual and Users Guide," DAIMI MD-30, Department of Computer Science University of Aarhus, Denmark (August 1979).

[42] Mosses, P. "Unified Algebras and Action Semantics," *Springer Lecture Notes in Computer Science 349* (1989), 17–35.

[43] Nielson, H.R., and Nielson, F. "Pragmatic Aspects of Two Level Denotational Meta-Languages," *Springer Lecture Notes in Computer Science 213* (1986), 133–143.

[44] Nielson, H.R., and Nielson, F. "Semantics Directed Compiling for Functional Languages," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming* (1986), 249–257.

[45] Oliva, D.P., and Wand, M. "A Verified Compiler for Pure PreScheme," Final Report for MITRE Contract Number F19628-89-C-001. Included in [15].

[46] Oliva, D.P., and Wand, M. "A Verified Runtime Structure for Pure PreScheme," Final Report for MITRE Contract Number F19628-89-C-001.

[47] Oliva, D.P. "The Linearization of Pure PreScheme Bytecode," Northeastern University College of Computer Science Technical Report, to appear.

[48] Oliva, D.P., Ramsdell, J.D., and Wand, M. "A Verified Compiler for VLISP PreScheme," submitted for publication.

[49] Paulson, L. "A Semantics-Directed Compiler Generator", *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages* (1982), 224–233.

[50] Peyton-Jones, S.L. "An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions," *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming* (August 1982), 150–158.

[51] Pierce, B.C. *Basic Category Theory for Computer Scientists*, MIT Press, Cambridge, MA, 1991.

[52] Plotkin, G.D. "Call-by-Name, Call-by-Value and the λ-Calculus," *Theoretical Computer Science 1* (1975) 125–159.

[53] Ramsdell, J.D. "The CURRY Chip," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming* (June 1986), 122–131.

[54] Ramsdell, J.D., Farmer, W.M., Guttman, J.D., Monk, L.G., and Swarup, V. "The VLISP PreScheme Front End," The MITRE Corporation Technical Report MTR92B098, October 1992.

[55] Raoult, J-C., and Sethi, R. "The Global Storage Needs of a Subcomputation," *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages* (1984), 148–157.

[56] Raskovsky, M.R. "Denotational Semantics as a Specification of Code Generators," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction* (1982), 264–275.

[57] Reynolds, J.C. *The Craft of Programming*, Prentice-Hall International, 1981.

[58] Schmidt, D.A. *Denotational Semantics: A Methodology for Language Development* Allyn and Bacon, Boston, 1986.

[59] Schonfinkel, M. "Uber die Blausteine der Mathematischen Logik," *Mathematische Annalen 92* (1924), 312.

[60] Sethi, R. "Control Flow Aspects of Semantics Directed Compiling," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction* (1982), 245–260.

[61] SPARC International, Inc. *The Sparc Architecture Manual — Version 8*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[62] Steele, G.L. "Rabbit: A Compiler for Scheme," MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[63] Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[64] Stoye, W.R., Clarke, T.J.W., and Norman, A.C. "Some Practical Methods for Rapid Combinator Reduction," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (August 1984), 159–166.

[65] Thatcher, J.W., Wagner, E.G., and Wright, J.B. "More Advice On Structuring Compilers and Proving The Correct," *Theoretical Computer Science 15* (1981), 223–250.

[66] Turner, D.A. "Another Algorithm for Bracket Abstraction," *The Journal of Symbolic Logic 44*, 2 (1979), 267–270.

[67] Turner, D.A. "A New Implementation Technique for Applicative Languages," *Software-Practice and Experience 9*, 1 (1979), 31–49.

[68] Volpano, D. "Software Templates," Ph.D. Thesis, The Oregon Graduate Center, 1986.

[69] Wall, D. "Global Register Allocation at Link Time," *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction* (1986), 264–275.

[70] Wall, D. "Register Windows vs. Register Allocation," *Proceedings of the ACM SIGPLAN '88 Symposium on Programming Language Design and Implementation* (1988), 67–78.

[71] Wand, M. "Semantics-Directed Machine Architecture," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages* (1982), 234–241.

[72] Wand, M. "Deriving Target Code as a Representation of Continuation Semantics," *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 496–517.

[73] Wand, M. "Loops in Combinator-Based Compilers," *Information and Control 57*, 2–3 (May/June 1983), 148–164.

[74] Wand, M. "A Semantic Prototyping System," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (June 1984), 213–221.

[75] Wand, M., and Montenyohl, M. "Correct Flow Analysis in Continuation Semantics," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages* (1988), 204–218.

[76] Wand, M. "A Short Proof of the Lexical Addressing Algorithm," *Information Processing Letters 35* (1990), 1–5.

[77] Wand, M., and Wang, Z.Y. "Conditional Lambda-Theories and the Verification of Static Properties of Programs," *Proc. 5th IEEE Symposium on Logic in Computer Science* (1990), 321–332.

[78] Wand, M. "Correctness of Procedure Representations in Higher-Order Assembly Language" *Proceedings Mathematical Foundations of Programming Semantics '91* (S. Brookes, ed.) *Springer Lecture Notes in Computer Science 598* (1992), 294–311.

[79] Wand, M., and Oliva, D.P. "Proving the Correctness of Storage Representations," *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming* (June 1992), 151–160.

[80] Wand, M. "Specifying the Correctness of Binding-Time Analysis," *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages* (1993), 137–143.