Sequent Calculus and Abstract Machines

Aaron Bohannon
University of Oregon

1 June 2004

Abstract

The sequent calculus proof system, as an alternative to natural deduction and Hilbert-style systems, has been well-appreciated for its structural symmetry and its applicability to automated proof search. Over the past decade, it has additionally been shown to have a computational significance, perhaps most notably as a type system for the $\overline{\lambda}$-calculus, designed by Hugo Herbelin. It is the purpose of this thesis to demonstrate that the structural properties of the $\overline{\lambda}$-calculus (derived from the underlying type system) are ideally suited for describing the states and transitions of several abstract machines that have been developed for executing programs written in a functional style. We explore the surrounding issues and offer formal translations of several abstract machines into versions of the $\overline{\lambda}$-calculus. In particular, we consider the Krivine machine, the SECD machine, and the ZINC abstract machine.

Acknowledgements

The basic idea of translating ZINC machine code and states into the $\overline{\lambda}$-calculus, along with the first sketch of a translation, was given to me by Hugo Herbelin. He guided me through earlier versions of translation and simulation. I would like to thank him for his guidance and his help in procuring the financial support of INRIA FUTURS in the summer of 2003.

I would very much like to thank my advisor at the University of Oregon, Zena Ariola, for being so helpful with my work on this thesis. She has been a great source of both theoretical and practical guidance and helped me to arrange my course schedule to make it possible to finish my thesis in a timely manner. Furthermore, she also played an essential role in arranging my summer support with INRIA in 2003.

I would also like to mention Peter Boothe, who talked over some ideas I had about machines and computation. Of course, any innacurracies or errors in this thesis are entirely my own.

Contents

Introduction

The study of computation is connected to the field of logic on many different levels. One of the most striking examples of this connection is the relationship known as the Curry-Howard isomorphism [12]. The core of this relationship is a correspondence between formal proofs in a logical inference system and terms of a programming language, and the most basic instance is the connection between intuitionistic natural deduction and the λ-calculus. Griffin has been extended it to classical natural deduction, showing that the corresponding language must contain operations to modify the control flow [8]. Curien and Herbelin extended the correspondence to sequent calculus in [4]. They also show that different ways of executing a program can be observed at the level of the logic. This thesis builds on their work. We show that sequent calculi naturally embed the data structures used in abstract machines and that these structures are a key element in decomposing the process of λ-term reduction to steps of minimal size. Therefore, we claim that sequent calculi are better suited than natural deduction for describing abstract machines.

The issues to be addressed in this thesis revolve around the observation that the order, manner, and size of the steps used to carry out a computation vary significantly depending upon what system is used. For instance, the traditional λ-calculus treats β-reduction as an operation that occurs in a single step, which would be considered a high-level operation relative to the capabilities of a computer modeled after a Turing machine. Non-determinism is another source of complexity, and reduction rules in most computational calculi are non-deterministic. Even with a deterministic reduction strategy, a single reduction step may make changes arbitrarily far from the top of the syntax tree, which has a computational overhead. Smaller factors also play a role in the complexity of operations, such as the use of symbolic names in contrast to the use of numbers.

Abstract machines are a tools for studying computation at a lower level, closer to the level at which physical machines work. They are useful both to those who study the theory and design of programming languages and to those who are interested in the practicalities of implementing languages. In particular, those who study functional programming languages based upon the λ-calculus have benefited from the development of abstract machines that can break up the rule of β-reduction into manageable components. Some of the earliest abstract machines designed for normalizing λ-terms were the Krivine machine and the SECD machine. The former is a simple but effective call-by-name machine designed by Jean-Louis Krivine.[1] The latter is a right-to-left call-by-value machine studied by G.D. Plotkin [18].

On the other hand, there has also been work directed at modifying the λ-calculus so that β-reduction can be simulated with smaller reduction steps. An important step in this direction was the investigation of explicit substitutions in [1]. Lescanne [16] offers a comparison of several versions of calculi with explicit substitutions. Curien, Hardin, and Lévy achieved an important goal in [3] by proving that the $\lambda\sigma_{\Uparrow}$-calculus is confluent on both closed and open terms. Furthermore, they introduce the notion of weak and strong calculi of explicit substitutions. In [9], a weak calculus of explicit substitution (the $\lambda\sigma_w$) is proposed as a useful "calculus of closures" for bridging the gap between abstract machines and the λ-calculus. They use it to prove the correctness of several abstract machines by developing translations from machine states to terms in the calculus.

This previously mentioned work on calculi with explicit substitutions was, generally speaking, motivated by the goal of deconstructing β-reduction into smaller steps. However, another calculus containing explicit substitutions was designed with an entirely different motivation. This was the $\overline{\lambda}$-calculus of Herbelin [10], which was conceived as a term assignment for sequent calculus proofs. The explicit substitutions in this calculus are present precisely for the purpose of encoding a particular proof structure. The subsequent work of Curien and Herbelin [4] of Wadler [21] in the area of sequent calculus proof terms has elucidated some of the symmetries inherent in computation, including the duality of the call-by-name and call-by-value reduction strategies.

There are now several versions of the $\overline{\lambda}$-calculus. The version in [4] does not contain explicit substitutions. (A more recent version of the $\overline{\lambda}$-calculus with explicit substitutions has been studied in [11].) However, one key and distinctive feature of all versions of the $\overline{\lambda}$-calculus is that application occurs between a term and an argument list, rather than a single argument as it would in the λ-calculus. On first consideration, it may seem as though this feature would make the calculus operate at a higher level, taking large steps that would need to be simulated by smaller steps in the λ-calculus. However, it turns out that almost the opposite is true. This will be investigated in Chapter 4.

The use of argument lists in the $\overline{\lambda}$-calculus actually grants a greater range of expressiveness in structuring terms than is available in the λ-calculus, and this more refined level of expressiveness is, in our opinion, an essential tool for capturing the low level details of β-reduction. We attempt to demonstrate this by using a version of the $\overline{\lambda}$-calculus to simulate the operation of three abstract machines. Not only do we simulate the operation of the abstract machines, but we show how this simulation can be done using a reduction strategy in the $\overline{\lambda}$-calculus for which the computational complexity of the $\overline{\lambda}$-term rewriting is constant.[2]

[1]More than 20 years after its informal introduction, the Krivine machine has not yet been officially presented in publication. Jean-Louis Krivine has recently written an article to remedy this situation [14].

[2]With a small exception in the case of the ZINC machine.

In the following three chapters of this thesis, we will begin by introducing various proof systems for minimal logic, the λ- and $\overline{\lambda}$-calculi, and abstract machines. Thereafter, we will present an enhanced version of the $\overline{\lambda}$-calculus, which we will call the $\overline{\lambda}\sigma_w$-calculus. Then we will use this calculus to simulate the Krivine abstract machine. In the final chapter, we will introduce the $\tilde{\mu}$ operator into the calculus and use the resulting $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus to simulate the execution steps of the SECD and ZINC abstract machines.

Chapter 1

Logic and Proof Terms

In this chapter, we will present natural deduction (ND) and the sequent calculus (both LJ [7] and LJT [5]) along with their corresponding term assignments. We will also review a few of the structural subtleties involved in these logical inference systems. Only the implicational fragment of minimal propositional logic will be considered. Although simple, this logic still has an important relationship with computation and will allow a cleaner presentation of the main points.

The syntax of the implicational fragment of minimal propositional logic is given in Figure 1.1. A formula is built from a set of atomic types (X), which we will leave unspecified, and a single logical connective ($\rightarrow$), which joins two logical formulas. We will use $A, B, C, \ldots$ as meta-variables that range over the set of formulas. For the semantics of the formulas, we will assume a *functional interpretation* of the propositional formulas (also known as Brouwer-Heyting-Kolmogorov semantics).

1.1 Natural Deduction

Although the sequent calculus proof system could be presented alone, we would like to present it in comparison with the more widely used system of natural deduction [20]. Natural deduction has become popular, especially in the area of computer science, for several reasons. One reason is that it has an important connection with the λ-calculus, which will be addressed shortly. This proof system is also popular due to the fact that its proofs can be read and constructed in a manner that is often considered more "natural" for humans than using

Figure 1.1: Syntax of the implicational fragment of minimal propositional logic

$$\boxed{A ::= X \mid A \rightarrow A}$$

Figure 1.2: Prawitz' natural deduction

$$\dfrac{A \to B \quad A}{B}\;\to_e \qquad \begin{array}{c}[A]^x\\ \vdots\\ \dfrac{B}{A \to B}\end{array}\;\to_i^x$$

Hilbert-style systems or sequent calculus.

1.1.1 Prawitz' Natural Deduction

The inference rules for Prawitz' version of natural deduction are given in Figure 1.2. There is one rule for the introduction of the implication connective and one for the elimination of the connective. Proofs correspond to trees where leaves represent the assumptions. A leaf can be *open* or *closed*. An open leaf would mean that the assumption is *active*. A closed leaf would correspond to an assumption that could have potentially been used in the proof but has been discharged by the end of it.

Whereas the elimination rule is fairly clear, since it corresponds to the traditional *modus ponens*, the notation of the implication introduction rule is fairly complex. In the elimination rule, the dots from the formula A to the formula B indicate a proof of B, which can refer to the assumption A zero or more times; the brackets indicate that, after the introduction of the connective, the assumption A may be discharged; and the variable x associates the use of this rule with the corresponding discharged assumption.

If one wants to know active assumptions at a point in the proof, one would need to travel up the tree to the leaves. To remedy this, we present natural deduction in sequent form. This also provides a more elegant way to clarify some details about using the implication introduction rule that have been left unspecified.

1.1.2 Natural Deduction in Sequent Form

A sequent is a syntactic construct for asserting a relation between propositions—in our case between a collection of formulas and a single formula—which we will write

$$\Gamma \vdash A$$

In this example, Γ is the *antecedent* and A is the *succedent*. When using sequents, it is necessary to specify what sort of "collection" the antecedent is. There are several possibilities: sets, mutlisets, sets of named formula, and sequences are the primary candidates.

A sequent is most naturally understood as a statement about derivability with respect to the Prawitz' form of natural deduction. $\Gamma \vdash A$ may be interpreted as, "There exists a proof tree whose conclusion is A, and all of whose

Figure 1.3: Natural deduction in sequent style

$$\frac{}{\Gamma, A \vdash A}\,Ax$$

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \rightarrow_e \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \rightarrow_i$$

open assumptions are contained within Γ." When interpreted as a statement about the existence of a proof, a sequent becomes a meta-proposition. This may be useful as a conceptual guide; however, the specification of a logical system using sequents is not, in general, formally dependent upon any previously-defined proof system.

Using sequents to formulate the rules of natural deduction allows clearer distinctions between the possible methods of managing assumptions. No longer must we work with open (or closed) assumptions at the leaves of the proof tree; instead, the leaves will contain an instance of an axiom in the inference system, and assumptions will be internalized into the antecedents of the sequents. Therefore, if one is interested in knowing the collection of active assumptions, one would simply look at the left-hand side of the sequent.

In reformulating natural deduction, the first step is to choose the nature of the collection. In order to note the differences, we will present three systems. The first is given in Figure 1.3. In this system, the antecedent of the sequents should be interpreted as a set of formulas.[1]

The correspondence with the previous form of natural deduction should be fairly clear. This system corresponds to a version of Prawitz' natural deduction in which:

1. a proof tree may have associated "leaves" that are not directly connected to its branches

2. implication introduction may not occur if there are no open assumptions of the formula associated to the tree

3. no discharge need actually take place

4. all equivalent open assumptions in the tree must be simultaneously discharged (and hence marked with the same name) if any of them are discharged

The first point arises from the fact that the axiom allows an arbitrary set of extra assumptions Γ. The second point is clear from the implication introduction rule. The third point can be observed by noting that, in the implication

[1]As commonly done, we will use the comma to indicate the union of two collections of formulas (as in Γ, Δ) or the union of a collection and a single formula (as in Γ, A). When the comma is used for sequences of assumptions, it should be interpreted as appending one to the other.

introduction rule, Γ may actually contain A, in which case $\Gamma, A = \Gamma$. The last point is inherent in the use of sets to maintain assumptions. Here is a simple proof in the system:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax}{A \vdash A \to A}\,\to_i}{\vdash A \to A \to A}\,\to_i$$

In the first implication introduction step, note how the assumption A is not deleted. If it were deleted, the second implication introduction step would not be possible.

If we decided to manage assumptions with multisets, then we could simply reinterpret the rules in Figure 1.3. In this new interpretation, the third and fourth statements above would be replaced with

3. exactly one open assumption in the tree must be discharged

The proof of $A \to A \to A$ in this system would be:

$$\dfrac{\dfrac{\dfrac{}{A, A \vdash A}\,Ax}{A \vdash A \to A}\,\to_i}{\vdash A \to A \to A}\,\to_i$$

Although these two versions of natural deduction in sequent form provide some clarification on the use of implication introduction, they both lack some expressive power with respect to Prawitz' version. Each of the proofs above could correspond to either of the two below:

$$\dfrac{\dfrac{[A]^x}{A \to A}\,\to_i^x}{A \to A \to A}\,\to_i^y \qquad \dfrac{\dfrac{[A]^x}{A \to A}\,\to_i^y}{A \to A \to A}\,\to_i^x$$

Using sets of named assumptions solves half of the problem. We could then tell which open assumption was discharged during an implication introduction, but would still not know which assumption corresponds to the succedent in the axiom rule. One solution to this would be to allow the attachment of names to the succedent in axiom rules. Another solution is simply to use sequences to manage assumptions. This requires a modification to the axiom, and the resulting system is presented in Figure 1.4. We will return to this system later on, when de Bruijn indices are discussed.

1.1.3 Structural Rules

When working with multisets or sequences in a proof system, it is sometimes necessary to add structural rules. The three structural rules typically considered are *weakening*, *contraction*, and *exchange* (Figure 1.5). Clearly, in a system that uses sets of assumptions, only weakening is relevant, and in a system that uses multisets, only weakening and contraction are relevant.

Figure 1.4: Natural deduction in sequent style with assumptions maintained as sequences

$$\frac{}{\Gamma, A, \Gamma' \ \vdash \ A}\ Ax$$

$$\frac{\Gamma \ \vdash \ A \rightarrow B \quad \Gamma \ \vdash \ A}{\Gamma \ \vdash \ B} \rightarrow_e \qquad \frac{\Gamma, A \ \vdash \ B}{\Gamma \ \vdash \ A \rightarrow B} \rightarrow_i$$

Figure 1.5: Structural rules

$$\frac{\Gamma \ \vdash \ B}{\Gamma, A \ \vdash \ B}\ Weak \qquad \frac{\Gamma, A, A \ \vdash \ B}{\Gamma, A \ \vdash \ B}\ Cont$$

$$\frac{\Gamma, A, \Gamma', B, \Gamma'' \ \vdash \ C}{\Gamma, B, \Gamma', A, \Gamma'' \ \vdash \ C}\ Exch$$

Adding these structural rules to a system may not alter the set of sequents that is provable in the system; in that case, the structural rules are *admissible* (but not *derivable*). This is the case for the three systems of natural deduction presented. It is worth noting that the precise formulation of the inference rules, especially the axiom, can influence the admissibility of various structural rules. For instance, if we had used the axiom

$$\frac{}{A \ \vdash \ A}\ Ax$$

in any of the inference systems, then weakening would be necessary to make the inference system complete and would no longer be an admissible rule.

1.2 Sequent Calculus

Having considered natural deduction, we now turn to the sequent calculus, which has been appreciated, since its introduction by Gentzen [7], for its symmetry and its applicability to automated proof search. Instead of having introduction and elimination rules as natural deduction does, the sequent calculus has right introduction rules and left introduction rules. Additionally, it has the significant *Cut* rule.

We first consider a sequent calculus proof system that manages assumptions as sets. The inference rules are given in Figure 1.6. This system is complete; all of the sequents that are derivable in the systems of natural deduction that we previously considered are also derivable in this system. Furthermore, it is complete without the cut rule; that is to say, the cut rule is admissible. (Demonstrating this fact is not trivial, as it would be to demonstrate the admissibility of weakening in one of our previous systems.)

Figure 1.6: Sequent calculus using sets of assumptions

$$\frac{}{\Gamma, A \vdash A}\,Ax$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\,\to_l \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,\to_r$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\,Cut$$

Remark 1.2.1. Even though the cut rule is admissible, its presence gives the sequent calculus a certain expressiveness that is quite valuable. For instance, it is relatively trivial to give a translation of natural deduction into sequent calculus when the cut rule is present. This expressiveness is one of the primary factors that makes the sequent calculus so useful a as basis for computational structures.

Now let us turn our attention to finding a version of the sequent calculus that manages its assumptions as multisets. In natural deduction, the presentation of the systems using sets and multisets required no typographical changes to the inference rules. Thus, we may be inclined to believe the same holds true with the sequent calculus. However, if we wish to maintain the property of cut admissibility, this is not the case. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{}{A \to B \vdash A \to B}\,Ax \quad \dfrac{}{A \to B, A \vdash A}\,Ax}{(A \to B) \to A, A \to B \vdash A}\,\to_l \quad \dfrac{}{(A \to B) \to A, A \to B, B \vdash B}\,Ax}{(A \to B) \to A, A \to B \vdash B}\,\to_l$$

This is a valid proof in the system of Figure 1.6 where assumptions are managed as sets. Note that in the left introduction step just before the conclusion, there is an implicit contraction step. In the application of the rule, we have

$$\Gamma = (A \to B) \to A, A \to B$$

However, the newly formed formula on the left is $A \to B$, which is already present as an assumption. So we get $\Gamma, A \to B = \Gamma$. There is a similar contraction in the first left introduction rule. These contractions would not be automatic if assumptions were kept as a multiset. There is no way to remedy the situation without changing the inference rules. We may either alter the left introduction rule or add a contraction rule to the system. We will take the latter choice, and the resulting system is presented in Figure 1.7.

Figure 1.7: Sequent calculus with multisets of assumptions

$$\frac{}{\Gamma, A \vdash A}\ Ax$$

$$\frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\ \to_l \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_r$$

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ Cont$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\ Cut$$

Figure 1.8: Syntax of λ-calculus terms

$$t \quad ::= \quad x \mid (\lambda x.t) \mid (t\ t')$$

1.3 Proof Terms

1.3.1 Natural Deduction

We would now like to present the λ-calculus as a means for encoding proofs. There is a a natural bijection between the terms of the basic λ-calculus and proofs in natural deduction. The syntax of the λ-calculus is presented in Figure 1.8, and the method for matching proofs to λ-terms is described by the inference rules in Figure 1.9. In particular, whereas λ-terms correspond to proofs, formulas correspond to types. This means that from now on, when we refer to a term calculus we actually mean a typed calculus à la Curry [2]. In this case, we are encoding proofs from a system that manages assumptions with sets of named formulas with a provision in the axiom to assign a name to the formula in the succedent of the sequent.

Once we can encode proofs as terms, it becomes much easier to work with transformations on proofs. The Curry-Howard isomorphism is properly an "isomorphism" because the reduction rules of the λ-calculus correspond to correct proof normalization steps. Reduction in the λ-calculus is done by means of the β-rule:

$$(\lambda x.t)\ u \to t[u/x]$$

Here we use the notation $t[u/x]$ as meta-syntax to describe the term t, with all free occurrences of x replaced by the term u.

1.3.2 Sequent Calculus

There have been many attempts to find proof terms for the sequent calculus proofs that have similar properties to those for natural deduction proofs. A

Figure 1.9: Assignment of λ-terms to proofs in natural deduction

$$\frac{}{\Gamma, x : A \vdash x : A}\,Ax$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B}\,\rightarrow_e \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x.t) : A \rightarrow B}\,\rightarrow_i$$

couple of problematic issues arise in this regard, though. In the case of natural deduction, we needed to include all the expressive structure of Prawitz' version into the sequents. This meant that names needed to be added to assumptions and to the succedent in the axiom. However, mimicking this in the sequent calculus is more difficult because the assumptions are manipulated by the left introduction rule. What name is to be given to the newly created formula at that point?

Another issue is that, although the *Cut* rule is admissible, there is often multiple cut-free proofs of the same sequent. This arises from the inherent permutability of the left- and right-introduction rules in the sequent calculus. We present the example from [10]:

$$\frac{\dfrac{\dfrac{}{A, C \vdash A}\,Ax \quad \dfrac{}{A, C, B \vdash B}\,Ax}{A \rightarrow B, A, C \vdash B}\,\rightarrow_l}{A \rightarrow B, A \vdash C \rightarrow B}\,\rightarrow_r \qquad \frac{\dfrac{}{A \vdash A}\,Ax \quad \dfrac{\dfrac{}{A, C, B \vdash B}\,Ax}{A, B \vdash C \rightarrow B}\,\rightarrow_r}{A \rightarrow B, A \vdash C \rightarrow B}\,\rightarrow_l$$

In proving this sequent, the right introduction rule that creates $C \rightarrow B$ and the left introduction rule that creates $A \rightarrow B$ are permutable. In some general sense, though, the two proof are the same. So, should these proofs have the same proof term or different ones?

The solution offered by Herbelin to both of these problems is the use of a restricted form of sequent calculus called *LJT* [5]. We will use two different types of sequents:

$$\Gamma \vdash A \qquad \Gamma \mid A \vdash B$$

Γ is still a collection of formulas, but now there is potentially a distinguished formula on the left, separated by the vertical bar. Now we may reformulate the rules of sequent calculus using both of these types of sequents. Figure 1.10 shows the rules other than cut. They are constructed so as to allow only one cut-free proof of any proposition. However, this new type of sequent offers four different ways of reformulating the cut rule (Figure 1.11). We will consider the system that uses all four. We will call the first two cut rules "head-cut" rules and the second two "mid-cut" rules.

1.3.3 The $\overline{\lambda}$-calculus

Finally, we may present a term language for representing proofs in this permutation-free version of the sequent calculus. Herbelin calls this the $\overline{\lambda}$-calculus, and

Figure 1.10: A version of sequent calculus with two types of sequents

$$\frac{}{\Gamma \mid A \vdash A}\ Ax$$

$$\frac{\Gamma \vdash A \quad \Gamma \mid B \vdash C}{\Gamma \mid A \rightarrow B \vdash C}\ \rightarrow_l \qquad \frac{\Gamma, A \vdash B}{\Gamma \mid A \vdash B}\ \rightarrow_r$$

$$\frac{\Gamma, A \mid A \vdash B}{\Gamma, A \vdash B}\ Cont$$

Figure 1.11: Four new types of cut

$$\frac{\Gamma \vdash A \quad \Gamma \mid A \vdash B}{\Gamma \vdash B}\ Cut_1$$

$$\frac{\Gamma \mid C \vdash A \quad \Gamma \mid A \vdash B}{\Gamma \mid C \vdash B}\ Cut_2$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\ Cut_3$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \mid C \vdash B}{\Gamma \mid C \vdash B}\ Cut_4$$

Figure 1.12: Syntax of $\overline{\lambda}$-terms

$$\begin{array}{rcl} t & ::= & x\,\{e\} \mid (\lambda x.t) \mid t\,\{e\} \mid t\,[\,s\,] \\ e & ::= & \Diamond \mid (t \cdot e) \mid (e \,@\, e') \mid e\,[\,s\,] \\ s & ::= & x := t \end{array}$$

Figure 1.13: Assignment of $\overline{\lambda}$-terms to proofs

$$\frac{}{\Gamma \mid .:A \ \vdash\ .\{\Diamond\}:A}\ Ax \qquad \frac{\Gamma, x:A \mid .:A \ \vdash\ .\{e\}:B}{\Gamma, x:A \ \vdash\ x\{e\}:B}\ Cont$$

$$\frac{\Gamma \ \vdash\ t:A \quad \Gamma \mid .:B \ \vdash\ .\{e\}:C}{\Gamma \mid .:A \to B \ \vdash\ .\{t \cdot e\}:C} \to_l \qquad \frac{\Gamma, x:A \ \vdash\ t:B}{\Gamma \ \vdash\ (\lambda x.t):A \to B} \to_r$$

$$\frac{\Gamma \ \vdash\ t:A \quad \Gamma \mid .:A \ \vdash\ .\{e\}:B}{\Gamma \ \vdash\ t\{e\}:B}\ Cut_1$$

$$\frac{\Gamma \mid .:C \ \vdash\ .\{e\}:A \quad \Gamma \mid .:A \ \vdash\ .\{e'\}:B}{\Gamma \mid .:C \ \vdash\ .\{e \mathbin{@} e'\}:B}\ Cut_2$$

$$\frac{\Gamma \ \vdash\ u:A \quad \Gamma, x:A \ \vdash\ t:B}{\Gamma \ \vdash\ t\,[x := u]:B}\ Cut_3$$

$$\frac{\Gamma \ \vdash\ u:A \quad \Gamma, x:A \mid .:C \ \vdash\ .\{e\}}{\Gamma \mid .:C \ \vdash\ .\{e\,[x := u]\}:B}\ Cut_4$$

its syntax is given in Figure 1.12. (We have made some very slight modifications to the original definition of the syntax to accommodate enhancements later on.) The syntax defines terms (t), contexts (e), and substitutions (s). Contexts are lists of arguments made from the constructors nil ($\Diamond$), cons ($\cdot$), and append (@), possibly with an associated substitution.[2] substitutions bind more tightly than the The terms will be used to represent proofs whose concluding sequent is of the form $\Gamma \ \vdash\ A$, and the contexts will represent proofs whose concluding sequent is of the form $\Gamma \mid A \ \vdash\ B$. In a sense, terms are producers of values whereas contexts are consumers. Substitutions will be used for recording the use of the mid-cut rules. The assignment of terms to proofs in the sequent calculus is given in Figure 1.13.

Herbelin [10] also offers reduction rules for the $\overline{\lambda}$-calculus (Figure 1.14)[3] and proves that these are strongly normalizing as a means for cut elimination. These rules offer a nice, orthogonal system for rewriting terms (*i.e.* proofs) [13]; however, they do not allow a simulation of β-reduction in the λ-calculus as one might hope. This will be necessary for our endeavor; so one of the next goals of this thesis is the development of a set of reduction rules for this term system

[2]We will assume that substitution binds more tightly than ($\cdot$) or (@) and omit parentheses when no ambiguity results.

[3]We must assume that all variable names in an expression are unique or provide some other means to avoid variable capture during substitution propagation.

Figure 1.14: Reduction rules of the $\overline{\lambda}$-calculus

(β_{nil})	$(\lambda x.t)\,\{\Diamond\}$	$\to$	$(\lambda x.t)$
(β_{cons})	$(\lambda x.t)\,\{u \cdot e\}$	$\to$	$t\,[\,x := u\,]\,\{e\}$
(C_{var})	$x\,\{e\}\,\{e'\}$	$\to$	$x\,\{e \mathbin{@} e'\}$
(C_{nil})	$\Diamond \mathbin{@} e$	$\to$	e
(C_{cons})	$(u \cdot e) \mathbin{@} e'$	$\to$	$u \cdot (e \mathbin{@} e')$
(S_{yes})	$x\,\{e\}\,[\,x := u\,]$	$\to$	$u\,\{e\,[\,x := u\,]\}$
(S_{no})	$y\,\{e\}\,[\,x := u\,]$	$\to$	$y\,\{e\,[\,x := u\,]\}$
(S_{λ})	$(\lambda y.t)\,[\,x := u\,]$	$\to$	$(\lambda y.t\,[\,x := u\,])$
(S_{nil})	$\Diamond\,[\,s\,]$	$\to$	$\Diamond$
(S_{cons})	$(t \cdot e)\,[\,s\,]$	$\to$	$t\,[\,s\,] \cdot e\,[\,s\,]$

that can simulate β-reduction.

Chapter 2

Weak Reduction and the $\lambda\sigma_w$-calculus

2.1 Reduction in the λ-calculus

The λ-calculus is an important theoretical model of computation. Moreover, it is also the archetypal minimal functional programming language. As a programming language, concerns arise with respect to implementation. The reduction relation induced by the β rule is not a functional relation. Since the λ-calculus is confluent, one possible implementation of reduction would be to follow all reduction paths simultaneously until one of them reaches a normal form. However, this is clearly unacceptable from a practical standpoint. Thus, we need to consider methods of reducing λ-terms that can be reasonably implemented.

2.1.1 Weak Reduction Strategies

An important factor related to efficiency is that *weak reduction* of λ-terms is often sufficient in programming languages. If we give an explicit specification of the reduction relation on λ-terms as in Figure 2.1, then we can define weak reduction by omitting the rule (ξ) from this inference system. In other words, reduction does not occur under abstractions. This is typically quite acceptable in most languages. Terms that cannot be reduced in this modified system are in *weak normal form.* However, as noted in [3, 9], weak normal forms are not unique, *i.e.* this reduction system is not confluent.

A *reduction strategy* is a deterministic (functional) subrelation of a general reduction relation. Since machines (under the typical model) operate deterministically, it is exponentially more efficient to specify a single reduction strategy than it is to simulate non-determinism with a deterministic machine. Furthermore, since the weak λ-calculus is not confluent, the latter is not even a useful option in a theoretical sense.

The three most basic weak reduction strategies are given in Figures 2.2, 2.3,

Figure 2.1: Definition of the reduction relation on λ-terms

$$(\lambda x.t)\ u \to t[u/x] \quad (\beta)$$

$$\frac{t \to t'}{\lambda x.t \to \lambda x.t'}\ (\xi) \qquad \frac{t \to t'}{t\ u \to t'\ u} \qquad \frac{u \to u'}{t\ u \to t\ u'}$$

Figure 2.2: Definition of call-by-name reduction

$$(\lambda x.t)\ u \to t[u/x] \qquad \frac{t \to t'}{t\ u \to t'\ u}$$

and 2.4. In the call-by-value reduction strategies, there is a constraint that a term be a *value*, which means that it cannot be further reduced by the strategy.

2.1.2 Closures

The previous alterations to the reduction relation were made with the goal of limiting the number of times that the β rule would need be applied in reducing a term. However, there is still a good deal of efficiency to be gained or lost in the implementation of the β rule itself. Implementing β-reduction primarily involves the propagation of substitutions through a term.

One important technique for efficient implementation is suspending the propagation of substitutions when possible. This is done by pairing an *environment* with a term to form a *closure*. The environment maps variable names either to terms or (more likely) to other closures. From a static point of view, a closure is a term with free variables that are defined in the associated environment. From a dynamic point of view, a closure is a term in which the propagation of substitutions has been suspended until it is clearly useful to propagate them further.

The process of suspending substitutions and working with closures cannot be described by the λ-calculus. Instead, we would need a calculus with more intensional properties—a sort of calculus of closures. Such a calculus could be useful for proving properties of mechanical implementations of λ-reductions. In [15], Leroy uses the λenv-calculus for such a purpose, and in [9], Hardin, Maranget, and Pagano propose the $\lambda\sigma_w$-calculus as ideally suited for this task.

Figure 2.3: Definition of left-to-right call-by-value reduction

$$(\lambda x.t)\ v \to t[v/x] \quad v \text{ is a value} \quad (\beta)$$

$$\frac{t \to t'}{t\ u \to t'\ u} \qquad \frac{u \to u' \quad v \text{ is a value}}{v\ u \to v\ u'}$$

Figure 2.4: Definition of right-to-left call-by-value reduction

$$(\lambda x.t)\ v \to t[v/x] \quad v \text{ is a value} \quad (\beta)$$

$$\frac{t \to t' \quad v \text{ is a value}}{t\ v \to t'\ v} \qquad \frac{u \to u'}{t\ u \to t\ u'}$$

Figure 2.5: Syntax of λ_{DB}-terms

$$t \quad ::= \quad \underline{n} \mid (\lambda t) \mid (t\ t')$$

It is a simplified version of the λenv that is only capable of weak reduction. Furthermore, the $\lambda\sigma_w$-calculus is confluent, thereby repairing, in a sense, the problem created by removing the ξ rule from the λ-calculus.

2.2 De Bruijn Indices

Before proceeding with a description of the $\lambda\sigma_w$-calculus, we will first introduce de Bruijn indices—a method of using natural numbers as the names of variables in λ-terms. Perhaps the best way to understand the motivation of Bruijn indices is to think about them from the perspective of logic. In the previous chapter, we introduced a method of maintaining assumptions in sequences rather than sets. If we would like to assign terms that can represent proofs in this system, we need to slightly modify the λ-calculus. We will call these new terms λ_{DB}-terms and their syntax is given in Figure 2.5. The assignment of terms to proofs (also the typing rules of λ_{DB}-terms) is given in Figure 2.6. (We use the notation $|\Gamma|$ here to represent the number of assumptions in the sequence Γ.) This typing system is essentially a fragment of a typing system presented in [1].

The benefit of de Bruijn indices is that they eliminate the extra effort of managing names for the assumptions. Of particular importance is the fact that they remove conflicts in names. The uniqueness of names (both locally and globally) is no longer something that must be enforced because it is now built into the naming scheme. Since a machine (abstract or physical) would naturally

Figure 2.6: Assignment of λ_{DB}-terms to proofs in natural deduction

$$\frac{(|\Delta| = n)}{\Gamma, A, \Delta\ \vdash\ \underline{n+1} : A}\ Ax$$

$$\frac{\Gamma\ \vdash\ t : A \to B \quad \Gamma\ \vdash\ u : A}{\Gamma\ \vdash\ (t\ u) : B} \to_e \qquad \frac{\Gamma, A\ \vdash\ t : B}{\Gamma\ \vdash\ (\lambda t) : A \to B} \to_i$$

Figure 2.7: Syntax of λ_{DB}-terms with explicit weakening

$$t \quad ::= \quad \bullet \mid (\lambda t) \mid (t\ t') \mid t\uparrow$$

Figure 2.8: Assignment of λ_{DB}-terms to proofs in natural deduction with explicit weakening

$$\frac{}{\Gamma, A\ \vdash\ \bullet : A}\ Ax$$

$$\frac{\Gamma\ \vdash\ t : A \rightarrow B \quad \Gamma\ \vdash\ u : A}{\Gamma\ \vdash\ (t\ u) : B}\ \rightarrow_e \qquad \frac{\Gamma, A\ \vdash\ t : B}{\Gamma\ \vdash\ (\lambda t) : A \rightarrow B}\ \rightarrow_i$$

$$\frac{\Gamma\ \vdash\ t : A}{\Gamma, B\ \vdash\ t\uparrow : A}\ Weak$$

use numbers to represent the infinite set of variable names, this simply provides an organized manner of doing so.

As mentioned by several authors (*e.g.* [16] and [3]), we don't actually need to depend upon an externally-defined set of natural numbers; instead, we can integrate them into our calculus if we have another term construction. Consider the new syntax in Figure 2.7. There are no natural numbers here. Instead we may define them:

$$\underline{n} := \bullet\ \overbrace{\uparrow \cdots \uparrow}^{n-1 \text{ times}}$$

The new terms correspond to a logic with an explicit weakening rule. These typing rules are given in Figure 2.8. We do not deal with reduction rules on this system, since we will presently discuss a method of reducing terms with de Bruijn indices in the $\lambda\sigma_w$-calculus.

2.3 The $\lambda\sigma_w$-calculus

The $\lambda\sigma_w$-calculus [3, 9] is a version of the λ-calculus with de Bruijn indices and simultaneous explicit substitutions. The syntax of $\lambda\sigma_w$-terms is given in Figure 2.9, and the reduction rules are given in Figure 2.10. Substitutions are now part

Figure 2.9: Syntax of $\lambda\sigma_w$-terms

$$\begin{array}{rcl} t & ::= & \underline{n} \mid \lambda t \mid (t\ t') \mid t[s] \\ s & ::= & \texttt{id} \mid t \cdot s \mid \uparrow \mid s \circ s' \end{array}$$

Figure 2.10: Reduction rules of the $\lambda\sigma_w$-calculus

(Beta)	$(\lambda t)\,[\,s\,]\,t'$	$\to$	$t\,[\,t' \cdot s\,]$
(App)	$(t\ t')\,[\,s\,]$	$\to$	$(t\,[\,s\,]\,t'\,[\,s\,])$
(FVar)	$\underline{1}\,[\,t \cdot s\,]$	$\to$	t
(RVar)	$\underline{n+1}\,[\,t \cdot s\,]$	$\to$	$\underline{n}\,[\,s\,]$
(Clos)	$(t\,[\,s\,])\,[\,s'\,]$	$\to$	$t\,[\,s \circ s'\,]$
(AssEnv)	$(s \circ s') \circ s''$	$\to$	$s \circ (s' \circ s'')$
(MapEnv)	$(t \cdot s) \circ s'$	$\to$	$t\,[\,s'\,] \cdot (s \circ s')$
(ShiftCons)	$\uparrow \circ\, (t \cdot s)$	$\to$	s
(IdL)	$\mathtt{id} \circ s$	$\to$	s

of the syntax. The substitution

$$[\,t \cdot u \cdot v \cdot \mathtt{id}\,]$$

means that t will be substituted for the variable $\underline{1}$, u will be substituted for the variable $\underline{2}$, and so on. This almost corresponds to

$$[\,\underline{1} := t\,]\,[\,\underline{2} := u\,]\,[\,\underline{3} := v\,]$$

in the $\overline{\lambda}$-calculus, except that the substitutions in the $\lambda\sigma_w$-calculus are *simultaneous*; so the substitution of u for $\underline{2}$ will never be applied to the term t in the $\lambda\sigma_w$-calculus as would eventually occur during reduction of the non-simultaneous substitutions in the $\overline{\lambda}$-calculus. The use of simultaneous substitutions necessitates a provision for substitution composition, which is found in the $\lambda\sigma_w$-calculus.

As mentioned before, one of the important features of this calculus is that this calculus natively implements weak β-reduction. In comparison with the $\overline{\lambda}$-calculus, this is possible because, among other things, substitutions can be propagated through applications. If we were to provide a typing system for this calculus, it would follow be similar to the basic one found in [1]. However, we will refrain from providing a typing system for a calculus with simultaneous explicit substitutions until we present the $\overline{\lambda}\sigma_w$-calculus.

Chapter 3

Abstract Machines

Broadly speaking, an *abstract machine* is a formal model of a computer. The key idea to be captured by an abstract machine is that, at any given point in time, a computer has a state, and, based upon the current state, the computer will progress to another state. Thus, an abstract machine is a *transition system* (as defined by Plotkin [19]). However, we will only be interested in deterministic abstract machines and will assume this to be an essential property. A set of final states may be described along with the transition system. If a machine cannot progress according to the transition relation, then it is either in a *final state* or it is *stuck*. If the set of final states is not explicitly given, then we assume that all states without a transition are valid final states.

A deterministic finite automaton and a Turing machine would both be examples of abstract machines,[1] but programming language researchers study abstract machines to figure out *how* computation is performed instead of simply *what* can be computed by various machines. This leads to abstract machines that look somewhat different from those used in automata theory.

Using abstract machines to study how computation can be performed is especially important with respect to functional programming languages because they operate in a manner very different from the natural behavior of physical machines. Another important use is a practical one: Implementing an abstract machine on multiple physical computer systems makes it possible to write just a single compiler for a language that outputs abstract machine instructions. An overview of the use of abstract machines with respect to programming languages can be found in [6].

[1]The *state* of a machine, in our terminology, will correspond to the concept of a machine *configuration* in automata theory. A set of *states*, in the terminology of automata theory would correspond to a set of atomic symbols that are used in defining the states of an abstract machine.

Figure 3.1: Syntax of arithmetic expressions

$$\begin{array}{rcl} e & ::= & n \mid (e+e) \\ n & ::= & \mathsf{0} \mid (n+1) \end{array}$$

3.1 Measuring Abstractness

The definition of "abstract machine" given so far allows for a large range in the level of abstractness. Abstractness arises when certain implementation details of the computational steps are ignored. Examples of abstract machines include the strategies for reducing λ-terms that were given in the previous chapter. (In that case the λ-term *is* the machine state.) However, these seem to be somewhat more "abstract" than we would expect of a typical machine. We will try to make this notion of abstractness more precise.

Since any useful machine will have an infinite number of potential states (*i.e.* "configurations"), the state transitions cannot be enumerated, but must be given by a set of rules. Certainly, using these rules to move the machine forward one step must be an effectively computable procedure; otherwise we have a machine that is not useful for studying the process of computation. But more practically, we can make a distinction between machines that can always be moved one step forward in constant time (*i.e.* $O(1)$) and those that cannot. It is important to note that this is as much a property of the data structure used to represent the machine state as it is of the rules governing the machine steps.[2]

If a machine's steps can always be executed in constant time then we will say that the machine takes *steps of constant size*, or steps of *strictly bounded* size. Sometimes a machine does not take constant-sized steps in a strict sense, but we can put a bound on the step size for all future states after statically examining the current state. If we can do this for all possible states, then we will say that the machine takes steps of *statically bounded* size. This is a much weaker notion than steps of strictly bounded size.

3.2 An Addition Machine

As a first example, we will consider two machines that evaluate simple arithmetic expressions on natural numbers (in unary representation). The syntax of the source language is given in Figure 3.1, and we include addition as the only operation. The first machine uses arithmetic expressions as its states, and its

[2]It is also dependent upon the model of computation being used. For our purposes, we will assume that the data structures being used are term trees and that manipulation of the branches of a tree within a fixed distance from the root can always be done in constant time. We will not assume the existence of random-access vectors by default, although, later on, we will mention how they may be important.

Figure 3.2: Arithmetic machine

$$\mathsf{0} + n \mapsto n \qquad (m+1) + n \mapsto m + (n+1)$$
$$\frac{e \mapsto e'}{n + e \mapsto n + e'} \qquad \frac{e \mapsto e'}{e + e'' \mapsto e' + e''}$$

transition relation is defined by the inference system in Figure 3.2. This machine could be called a small-step operational semantics on the language [19]. In fact, any small-step operational semantics really defines an abstract machine. Furthermore, any strategy in a calculus can be called a "machine" by this definition. However, this addition machine does not have constant sized steps. In general, if it is necessary to use premises in an inference system in order to define a machine's transition rules, then the machine does not take constant-sized steps. (There are various ways in which the converse may fail to be true, though.)

Next, we present another machine that operates at a lower level and does take constant-sized steps. This machine will work with a control stack and a value stack. A control stack (C) is a list of expressions or an Add instruction. A value stack (S) is a list of integers.

$$\begin{array}{rcl} C & ::= & \varepsilon \mid \mathsf{Add};\ C \mid e;\ C \\ S & ::= & \varepsilon \mid n \cdot S \end{array}$$

A machine state is simply a pair consisting of the control stack and the value stack:

$$M \quad ::= \quad (C\ /\ S)$$

The transition rules of the machine are:

$$\begin{array}{rcl} (n;\ C\ /\ S) & \mapsto & (C\ /\ n \cdot S) \\ ((e + e');\ C\ /\ S) & \mapsto & (e;\ e';\ \mathsf{Add};\ C\ /\ S) \\ (\mathsf{Add};\ C\ /\ \mathsf{0} \cdot n \cdot S) & \mapsto & (C\ /\ n \cdot S) \\ (\mathsf{Add};\ C\ /\ (m+1) \cdot n \cdot S) & \mapsto & (C\ /\ m \cdot (n+1) \cdot S) \end{array}$$

Now we need a means configure the starting state of the machine. A function ($\mathcal{L}$) to *load* an expression can be defined as:

$$\mathcal{L}(e) = (e;\ /\ \varepsilon)$$

We may also want a function to *unload* a value from the machine after it has stopped. This simply takes the result out of the machine. In this case, it will suffice to say that the result is the item on top of the value stack when the control stack is empty.

3.3 Compilation and Decompilation

In both machines given above, the source expression was put directly into the machine. Often, though, an abstract machine is designed to operate on a source expression that has been *compiled*. (The notion that programs are compiled into a list of instructions is sometimes useful, but except for very simple languages, the instructions in compiled code will often refer to other pieces of compiled code, in which case the actual structure of the code bears more resemblance to a tree or graph rather than a list.)

As an example of compilation, here is a function that will compile expressions for a modified version of the arithmetic machine.

$$\begin{array}{rcl} [\![\underline{n}]\!] & = & \underline{n} \\ [\![e+e']\!] & = & [\![e]\!];\ [\![e']\!];\ \mathsf{Add} \end{array}$$

It is easy to see that this function just turns the source expression into postfix notation, which is a typical thing to do during compilation. Here is the modified machine that will execute compiled arithmetic expressions.

$$\begin{array}{rcl} (n\ ;\ C\ /\ S) & \mapsto & (C\ /\ n\cdot S) \\ (\mathsf{Add};\ C\ /\ \mathsf{0}\cdot n\cdot S) & \mapsto & (C\ /\ n\cdot S) \\ (\mathsf{Add};\ C\ /\ (m+1)\cdot n\cdot S) & \mapsto & (C\ /\ m\cdot(n+1)\cdot S) \end{array}$$

Finally, we may provide a compile-and-load function ($\mathcal{L}$) for running the machine on an expression:

$$\mathcal{L}(e) = ([\![e]\!];\ /\ \varepsilon)$$

Now we may compare the machine that executes compiled code with the one that operates on source expressions. The difference is that exactly one rule has been removed. This raises the question as to whether this is really a different machine or not. In essence, the original machine integrated a compilation of terms on-the-fly during execution. When we present the SECD machine later, we consider a version that operates on compiled code instead of source terms as the original SECD machine did [18]. The changes made to the SECD machine correspond exactly to the changes made to this arithmetic machine.

Decompilation indicates the translation of code, or an entire machine state to an output language. If defined on machine states, it will usually be a total function, and not just restricted to final states (as unloading may be). In the later chapters, we will define several decompilation functions from abstract machine states to a version of the $\overline{\lambda}$-calculus that will be presented in the next chapter.

Chapter 4

A Calculus for Simulating Abstract Machines

As discussed in Chapter 2, if we are to find a calculus that operates at the same level as an abstract machine, then it should be able to perform weak reduction and to represent and manage closures. The $\lambda\sigma_w$-calculus satisfies these conditions, but there is another factor that we cover presently.

4.1 The Structure of Applicative Terms

Consider the application of a term to three arguments in the λ-calculus and the $\overline{\lambda}$-calculus:

$$t\ u_1\ u_2\ u_3 \qquad\qquad t\,\{u_1 \cdot u_2 \cdot u_3 \cdot \Diamond\}$$

While the expressions look somewhat similar when read on paper, their abstract syntax trees are quite distinct. Compare the two trees in Figure 4.1. In the case of the λ-term, the subterm t is at the bottom of the tree, and in the case of the $\overline{\lambda}$-term, it is at the top. This is very significant because, in the definitions of all the useful reduction strategies for the λ-calculus, we find a rule of the form:

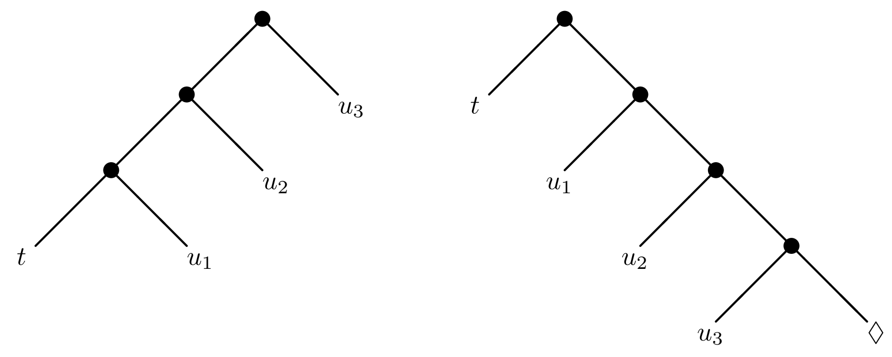
$$\frac{t \rightarrow t'}{t\ u \rightarrow t'\ u}$$

An implementation of reduction based upon this rule requires a search of arbitrary depth into the λ-term. Such a search would also be required simply to discover whether the term has a redex at all. However, there is no way to avoid this situation when working directly with λ-terms—it is a necessity given their structure.

The applicative $\overline{\lambda}$-term, on the other hand, always has a redex at the top of the term. If t is an abstraction (*e.g.* $t = (\lambda x.t')$), then we have:

$$(\lambda x.t')\,\{u_1 \cdot u_2 \cdot u_3 \cdot \Diamond\} \quad\rightarrow\quad t'\,[\,x := u_1\,]\,\{u_2 \cdot u_3 \cdot \Diamond\}$$

Figure 4.1: The structure of applicative terms in the λ-calculus and the $\overline{\lambda}$-calculus



And if t is another application (*e.g.* $t = t' \{e\}$), then we have:

$$t' \{e\} \{u_1 \cdot u_2 \cdot u_3 \cdot \Diamond\} \quad \rightarrow \quad t' \{e \mathbin{@} (u_1 \cdot u_2 \cdot u_3 \cdot \Diamond)\}$$

The case where $t = x \{e\}$ is effectively the same, and in the case where $t = t' [\, s\,]$, the substitution can be propagated. Hence, in an applicative term in the $\overline{\lambda}$-calculus, *there is always a redex within a fixed distance from the top of the syntax tree.* This property is inherited from the process of cut reduction in the sequent calculus LJT. This is not the case in the λ-calculus nor in the $\lambda\sigma_w$-calculus, both of whose terms are related to proofs in natural deduction.

This property is of central importance in the design of a calculus that can simulate abstract machines. The $\overline{\lambda}$-calculus becomes even more useful when we realize that there is also a means to express the applicative structure of the λ-calculus with $\overline{\lambda}$-terms. This is pointed out by Herbelin in [10]. Consider the syntax trees of the following two expressions:

$$t\ u_1\ u_2\ u_3 \qquad\qquad t \{u_1 \cdot \Diamond\} \{u_2 \cdot \Diamond\} \{u_3 \cdot \Diamond\}$$

The translation from the first structure to the second is straightforward and actually represents a translation from a proof in natural deduction to one in the sequent calculus. As noted by Herbelin, the $\overline{\lambda}$-term reduces to the form

$$t \{u_1 \cdot u_2 \cdot u_3 \cdot \Diamond\}$$

Moreover, it is also important to note that it can make this transformation with a series of reduction steps that takes place entirely within a fixed distance from the top of the syntax tree. The reduction is shown in Figure 4.2. The ability of the $\overline{\lambda}$-calculus to express application in two ways is really the same expressiveness that was noted in Chapter 1 with respect to the sequent calculus.

Figure 4.2: Normalization of a natural-deduction style applicative term in the $\overline{\lambda}$-calculus

$$\begin{array}{ll}
 & t\,\{u_1 \cdot \Diamond\}\,\{u_2 \cdot \Diamond\}\,\{u_3 \cdot \Diamond\} \\
\to & t\,\{u_1 \cdot \Diamond\}\,\{(u_2 \cdot \Diamond) \mathbin{@} (u_3 \cdot \Diamond)\} \\
\to & t\,\{u_1 \cdot \Diamond\}\,\{u_2 \cdot (\Diamond \mathbin{@} (u_3 \cdot \Diamond))\} \\
\to & t\,\{u_1 \cdot \Diamond\}\,\{u_2 \cdot u_3 \cdot \Diamond\} \\
\to & t\,\{(u_1 \cdot \Diamond) \mathbin{@} (u_2 \cdot u_3 \cdot \Diamond)\} \\
\to & t\,\{u_1 \cdot (\Diamond \mathbin{@} (u_2 \cdot u_3 \cdot \Diamond))\} \\
\to & t\,\{u_1 \cdot u_2 \cdot u_3 \cdot \Diamond\}
\end{array}$$

4.2 Merging $\overline{\lambda}$ and $\lambda\sigma_w$

4.2.1 Simultaneous Substitutions

After observing the structures of applicative terms in the λ- and $\overline{\lambda}$-calculi, we must turn attention to the structures of terms with explicit substitutions. We have already described the use of simultaneous substitutions in the $\lambda\sigma_w$-calculus, so let's present a term from the $\overline{\lambda}$-calculus and the corresponding one from the $\lambda\sigma_w$-calculus side by side:

$$t\,[\,x := u_1\,]\,[\,y := u_2\,]\,[\,z := u_3\,] \qquad\qquad t\,[\,u_1 \cdot u_2 \cdot u_3 \cdot \texttt{id}\,]$$

Ignoring the differences in meaning and the fact that only one uses variable names, we can note that the structures of these two terms are completely analogous with the structures observed in the previous section. In this case, though, it is the $\overline{\lambda}$-calculus whose potential redex is buried within the syntax tree. A redex for the $\lambda\sigma_w$-calculus would exist at the top of its term.

4.2.2 Weak Reduction

With respect to the operations of abstract machines that implement reduction on λ-terms, the $\lambda\sigma_w$-calculus surpasses the $\overline{\lambda}$-calculus in another way: the $\overline{\lambda}$-calculus has no direct support for closures or weak reduction. Moreover, its treatment of substitution propagation prevents it from even simulating the β-reduction of the λ-calculus, as noted by Herbelin. Thus, we present a new calculus with the intention of bringing together the desirable traits of the $\overline{\lambda}$-calculus and the $\lambda\sigma_w$-calculus.

4.3 The $\overline{\lambda}\sigma_w$-calculus

The $\overline{\lambda}\sigma_w$-calculus is a calculus with λ-abstractions, argument lists, simultaneous explicit substitutions, de Bruijn indices, and a form of weak β-reduction.[1] Its

[1]Herbelin [10] describes a version of the $\overline{\lambda}$-calculus with de Bruijn indices and explicit weakening, also. However, it did not have simultaneous substitutions, and its reduction rules

Figure 4.3: Syntax of the $\overline{\lambda}\sigma_w$-calculus

$$
\begin{array}{rcl}
t & ::= & \bullet\{e\} \mid \lambda t \mid t\{e\} \mid t[s] \\
e & ::= & \Diamond \mid (t \cdot e) \mid (e \mathbin{@} e') \mid e[s] \\
s & ::= & \texttt{id} \mid (t \cdot s) \mid (s \circ s') \mid \uparrow
\end{array}
$$

syntax is given in Figure 4.3. As in the $\overline{\lambda}$-calculus, there are terms (t), contexts (e), and substitutions (s). The primary changes in the syntax are the removal of variable names from terms, and the expanded constructs for substitutions. There is now only one variable name ($\bullet$), which is similar to a de Bruijn index of `1`. We will define our indices in this calculus in terms of this base variable and the weakening substitution via the following abbreviations:

$$
\begin{array}{rcl}
[\uparrow]^n & := & \overbrace{[\uparrow]\cdots[\uparrow]}^{n \text{ times}} \\
\underline{n} & := & \bullet\{\Diamond\}[\uparrow]^{n-1}
\end{array}
$$

We also provide typing rules (*i.e.* a proof system) for the terms of this calculus. They are given in Figure 4.4. The typing of substitutions is based on the examples in [1] and [11]. We present the reduction rules for the $\overline{\lambda}\sigma_w$-calculus in Figure 4.5. These are somewhat different from the reduction rules of the $\overline{\lambda}$-calculus. They are instead modeled after the reduction rules of the $\lambda\sigma_w$-calculus. The key features are that substitutions can no longer be propagated under λ-abstractions and they can be propagated through applications. We conjecture that this calculus shares the confluence properties of the $\lambda\sigma_w$-calculus.

We will now use this calculus to simulate the reduction steps of the Krivine machine, as the first example of its correspondence to abstract machines.

4.4 Simulating the Krivine Machine

The Krivine machine is a simple machine for reducing λ-terms according to the call-by-name strategy [14]. It has been described many times. One correctness proof can be found in [22]. Curien and Herbelin [4] describe a version that operates at a higher-level, performing substitution in one step and thereby eliminating the closures managed by most versions, but their example of the Krivine machine offered a clear motivation for the use of applicative contexts in the $\overline{\lambda}$-calculus. In this chapter, we will try to extend the correspondence to a machine that does maintain closures and our $\lambda\sigma_w$-calculus, which supports reduction on closures.

were equivalent to those of the original $\overline{\lambda}$-calculus.

Figure 4.4: Assignment of types to $\overline{\lambda}\sigma_w$-terms

$$\frac{}{\Gamma \mid .:A \vdash .\{\Diamond\}:A} \qquad \frac{\Gamma, A \mid .:A \vdash .\{e\}:B}{\Gamma, A \vdash \bullet\{e\}:B}$$

$$\frac{\Gamma \vdash t:A \qquad \Gamma \mid .:B \vdash .\{e\}:C}{\Gamma \mid .:A \to B \vdash .\{t \cdot e\}:C} \qquad \frac{\Gamma, A \vdash t:B}{\Gamma \vdash \lambda t: A \to B}$$

$$\frac{\Gamma \mid .:C \vdash .\{e\}:A \qquad \Gamma \mid .:A \vdash .\{e'\}:B}{\Gamma \mid .:C \vdash .\{e \,@\, e'\}:B}$$

$$\frac{\Gamma \vdash t:A \qquad \Gamma \mid .:A \vdash .\{e\}:B}{\Gamma \vdash t\{e\}:B}$$

$$\frac{s:(\Gamma \Rightarrow \Delta) \qquad \Gamma \mid .:A \vdash .\{e\}:B}{\Delta \mid .:A \vdash .\{e[s]\}:B} \qquad \frac{s:(\Gamma \Rightarrow \Delta) \qquad \Gamma \vdash t:A}{\Delta \vdash t[s]:A}$$

$$\frac{}{\texttt{id}:(\Gamma \Rightarrow \Gamma)}$$

$$\frac{}{\uparrow:(\Gamma \Rightarrow \Gamma, A)}$$

$$\frac{s:(\Gamma \Rightarrow \Delta) \qquad \Delta \vdash t:A}{t \cdot s:(\Gamma, A \Rightarrow \Delta)}$$

$$\frac{s:(\Gamma \Rightarrow \Delta) \qquad s':(\Delta \Rightarrow \Sigma)}{s \circ s':(\Gamma \Rightarrow \Sigma)}$$

Figure 4.5: Reduction rules of the $\overline{\lambda}\sigma_w$-calculus

(BetaNil)	$(\lambda t)\,[\,s\,]\,\{\Diamond\}$	$\rightarrow$	$(\lambda t)\,[\,s\,]$
(BetaCons)	$(\lambda t)\,[\,s\,]\,\{t' \cdot e\}$	$\rightarrow$	$t\,[\,t' \cdot s\,]\,\{e\}$
(AppApp)	$t\,\{e\}\,\{e'\}$	$\rightarrow$	$t\,\{e \mathbin{@} e'\}$
(ConcatNil)	$\Diamond \mathbin{@} e$	$\rightarrow$	e
(ConcatCons)	$(t \cdot e) \mathbin{@} e'$	$\rightarrow$	$t \cdot (e \mathbin{@} e')$
(ConcatAssoc)	$(e \mathbin{@} e') \mathbin{@} e''$	$\rightarrow$	$e \mathbin{@} (e' \mathbin{@} e'')$
(SubVarNil)	$\bullet\,\{\Diamond\}\,[\,t \cdot s\,]$	$\rightarrow$	t
(SubVarCons)	$\bullet\,\{e\}\,[\,t \cdot s\,]$	$\rightarrow$	$t\,\{e\,[\,t \cdot s\,]\}$
(SubApp)	$t\,\{e\}\,[\,s\,]$	$\rightarrow$	$t\,[\,s\,]\,\{e\,[\,s\,]\}$
(SubSubTerm)	$t\,[\,s\,]\,[\,s'\,]$	$\rightarrow$	$t\,[\,s \circ s'\,]$
(SubNil)	$\Diamond\,[\,s\,]$	$\rightarrow$	$\Diamond$
(SubCons)	$(t \cdot e)\,[\,s\,]$	$\rightarrow$	$t\,[\,s\,] \cdot e\,[\,s\,]$
(SubSubContext)	$e\,[\,s\,]\,[\,s'\,]$	$\rightarrow$	$e\,[\,s \circ s'\,]$
(SubConcat)	$(e \mathbin{@} e')\,[\,s\,]$	$\rightarrow$	$e\,[\,s\,] \mathbin{@} e'\,[\,s\,]$
(CompNil)	$\mathtt{id} \circ s$	$\rightarrow$	s
(CompCons)	$(t \cdot s) \circ s'$	$\rightarrow$	$t[s'] \cdot (s \circ s')$
(CompShift)	$\uparrow \circ\, (t \cdot s)$	$\rightarrow$	s
(CompAssoc)	$(s \circ s') \circ s''$	$\rightarrow$	$s \circ (s' \circ s'')$

4.4.1 Definition of the Machine

Our definition of the Krivine machine closely follows that of [9]. The machine code comprises three instructions:

$$\begin{array}{rcl} C & ::= & \mathsf{Acc}(n) \\ & \mid & \mathsf{Grab};\ C \\ & \mid & \mathsf{Push}(C);\ C' \end{array}$$

The basic values[2] that are manipulated by the machine are closures. In this case, a closure is any code that is paired with an environment, an environment simply being a list of values.

$$\begin{array}{rcl} V & ::= & (C/E) \\ E & ::= & \varepsilon \mid V.E \end{array}$$

During execution of the code, the machine will manipulate a stack of values, which we will call the argument stack:

$$A \quad ::= \quad \varepsilon \mid V.A$$

[2]Calling these closures "values" with respect to the machine is traditional, although they are not necessarily values with respect to the call-by-name strategy.

Figure 4.6: Rules for the Krivine abstract machine

$$\boxed{\begin{array}{rcl}(\mathsf{Acc}(1)\ /\ (C/E).E'\ /\ A) & \mapsto & (C\ /\ E\ /\ A)\\ (\mathsf{Acc}(n+1)\ /\ V.E\ /\ A) & \mapsto & (\mathsf{Acc}(n)\ /\ E\ /\ A)\\ (\mathsf{Grab};\ C\ /\ E\ /\ V.A) & \mapsto & (C\ /\ V.E\ /\ A)\\ (\mathsf{Push}(C);\ C'\ /\ E\ /\ A) & \mapsto & (C'\ /\ E\ /\ (C/E).A)\end{array}}$$

The total state of the machine is represented by three components: the current code, the current environment, and the current argument stack:

$$M \quad ::= \quad (C\ /\ E\ /\ A)$$

The machine takes steps according to the transition rules in Figure 4.6. By examining these transitions, we can see exactly two cases in which the machine cannot progress. One case occurs when there is an Acc instruction but the environment is empty. We will say that the machine is stuck in this case. This cannot occur when running the machine on an expression without free variables. The other case occurs when there is a Grab instruction but the argument stack is empty. In this case, we are at a final state.

4.4.2 Compiling and Loading

This machine will use λ_{DB}-terms as its basic source language, in which case the compilation function is quite trivial. (A more complex function could compile λ-terms with named variables.) When viewed as unlabeled trees the source term and the compiled code are isomorphic.

$$\begin{array}{rcl}[\![\underline{n}]\!] & = & \mathsf{Acc}(n)\\ [\![\lambda t]\!] & = & \mathsf{Grab};\ [\![t]\!]\\ [\![(t\ u)]\!] & = & \mathsf{Push}([\![u]\!]);\ [\![t]\!]\end{array}$$

The compile-and-load function $\mathcal{L}$ simply puts a compiled version of the λ-term into a machine with an empty environment and argument stack.

$$\mathcal{L}(t) \quad = \quad ([\![t]\!]\ /\ \varepsilon\ /\ \varepsilon)$$

After running the machine, we can unload the code (which begins with Grab) and the environment as we wish. (The argument stack must be empty.)

4.4.3 Decompiling to the $\overline{\lambda}\sigma_w$-calculus

It is interesting to note that although the Krivine machine structure can be easily defined by an inductive definition as it is above, it is typical to describe decompilation functions that do not operate on this natural structure, but instead treat the argument stack as a sequence [9, 22]. Some difficulty arises from the fact that the structure of the argument stack is contrary to the natural structure of λ-terms. Yet, it is exactly the structure of the argument stack that allows the Krivine machine to progress smoothly with constant-sized steps. However, if we consider a decompilation to the $\overline{\lambda}$-calculus, this issue with the structure of the argument is immediately resolved.

We now present a decompilation function from the states of the Krivine machine to the $\overline{\lambda}\sigma_w$-calculus and then prove that execution can be simulated without loss of detail. The decompilation will be purely compositional following the natural syntactic structure of the machines components. First, we define how to decompile machine code (decompilation of code is denoted $\langle\!\langle C \rangle\!\rangle$):

$$\begin{array}{rcl} \langle\!\langle \mathsf{Acc}(n) \rangle\!\rangle & = & \underline{n} \\ \langle\!\langle \mathsf{Grab};\ c \rangle\!\rangle & = & \lambda\, \langle\!\langle c \rangle\!\rangle \\ \langle\!\langle \mathsf{Push}(c);\ c' \rangle\!\rangle & = & \langle\!\langle c' \rangle\!\rangle \,\{\langle\!\langle c \rangle\!\rangle \cdot \Diamond\} \end{array}$$

We will denote the decompilation function for closures, environments, argument stacks, and machine states as $\overline{M}$, using a horizontal bar.[3] Closures have a straightforward decompilation.

$$\overline{(C/E)} \quad = \quad \langle\!\langle C \rangle\!\rangle \,[\,\overline{E}\,]$$

Environments are also easily decompiled.

$$\begin{array}{rcl} \overline{\varepsilon} & = & \mathtt{id} \\ \overline{V.E} & = & \overline{V} \cdot \overline{E} \end{array}$$

The decompilation of argument stacks is similarly compositional.

$$\begin{array}{rcl} \overline{\varepsilon} & = & \Diamond \\ \overline{V.E} & = & \overline{V} \cdot \overline{E} \end{array}$$

And finally the decompilation of machine states simply puts these pieces together.

$$\overline{(C\ /\ E\ /\ A)} \quad = \quad \langle\!\langle C \rangle\!\rangle \,[\,\overline{E}\,]\,\{\overline{A}\}$$

Remark 4.4.1. Note that this translation is an injection. Hence, a partial function for re-compilation could be defined from $\overline{\lambda}\sigma_w$-terms to Krivine machine states.

4.4.4 The Krivine Strategy in the $\overline{\lambda}\sigma_w$-calculus

Now that we can translate states of the Krivine machine into the $\lambda\sigma_w$-calculus, we would hope that machine steps correspond to reduction steps in the calculus. Not only can we demonstrate this, but we can demonstrate a stronger proposition:

[3] We would also have liked to use the more standard notation of a horizontal bar for the decompilation of code, but later on, we will need to add parameters to the decompilation function. Thus, the notation $\langle\!\langle C \rangle\!\rangle$ will be more suitable.

Figure 4.7: The V strategy

Using rightmost outermost priority:

$$t\,[\,\uparrow \circ (t' \cdot s)\,] \xrightarrow{V} t\,[\,s\,]$$

$$t\,[\,s\,]\,[\,s'\,] \xrightarrow{V} t\,[\,s \circ s'\,] \qquad t\,\{e\}\,[\,s\,] \xrightarrow{V} t\,[\,s\,]\,\{e\,[\,s\,]\}$$

$$\bullet\,\{\Diamond\}\,[\,t \cdot s\,] \xrightarrow{V} t$$

Figure 4.8: The P strategy

$$\Diamond\,[\,s\,] \xrightarrow{P} \Diamond \qquad (t \cdot e)\,[\,s\,] \xrightarrow{P} t\,[\,s\,] \cdot e\,[\,s\,] \qquad t \cdot \Diamond\,[\,s\,] \xrightarrow{P} t \cdot \Diamond$$

Proposition 4.4.2. *There exists a deterministic strategy K in the $\overline{\lambda}\sigma_w$-calculus such that*

a. the reduction steps performed by K are confined to a fixed distance from the top of the syntax tree

b. if M is a final state in the Krivine machine, then $\overline{M}$ is a K-value

c. if $M \mapsto M'$ in the Krivine machine, then $\overline{M} \xrightarrow{K}{}^{n} \overline{M'}$, for some constant n

First we present the strategy. We decompose the strategy K into sub-strategies to make the presentation easier, but it is crucial to note that there are no inference rules that create a recursive definition of a relation. Hence, all reductions specified by the strategy are confined to a fixed distance from the top of the syntax tree.

The V strategy is shown in Figure 4.7 and propagates substitutions through terms. A priority must be assigned among these rules in order to make the strategy deterministic. We have visually presented the rules so that rules toward

Figure 4.9: The J strategy

$$\frac{e \xrightarrow{P} e'}{e \mathbin{@} e'' \xrightarrow{J} e' \mathbin{@} e''}$$

$$\Diamond \mathbin{@} e \xrightarrow{J} e \qquad (t \cdot e) \mathbin{@} e' \xrightarrow{J} t \cdot (e \mathbin{@} e') \qquad t \cdot (\Diamond \mathbin{@} e) \xrightarrow{J} t \cdot e$$

Figure 4.10: The K strategy

Using rightmost outermost priority:

$$t\,\{e\}\,\{e'\} \xrightarrow{K} t\,\{e\,@\,e'\}$$

$$\frac{e \xrightarrow{J} e'}{t\,\{e\} \xrightarrow{K} t\,\{e'\}}$$

$$\frac{t \xrightarrow{V} t'}{t\,\{e\} \xrightarrow{K} t'\,\{e\}} \qquad (\lambda t)\,[\,s\,]\,\{t' \cdot e\} \xrightarrow{K} t\,[\,t' \cdot s\,]\,\{e\}$$

the top of the page have higher priority; however, it should be clear if we specify that the changes made by these rules must occur in a *rightmost outermost* manner.

The P strategy is shown in Figure 4.8 and propagates substitutions though contexts. The J strategy propagates context concatenation and is shown in Figure 4.9. The rules of the J strategy do not overlap, although an examination of the P strategy is necessary to confirm this. Finally, the K strategy is shown in Figure 4.10. Its rules do overlap, and again, we choose a *rightmost outermost* priority.

4.4.5 Simulation

Part *a.* of Proposition 4.4.2 can thus be verified by examination of the strategy given. To confirm part *b.*, let

$$M = (\mathsf{Grab};\ C\ /\ E\ /\ \varepsilon)$$

Then,

$$\overline{M} = (\lambda\,\langle\!\langle C \rangle\!\rangle)\,\left[\,\overline{E}\,\right]\,\{\Diamond\}$$

which is irreducible by the strategy K.

To confirm part *c.*, we now provide the specific steps in the $\overline{\lambda}\sigma_w$-calculus that simulate the steps of the Krivine machine. The choice of the Krivine machine rule is determined by the current instruction; so we consider cases according to the next instruction to be executed.

- Assume the next instruction is $\mathsf{Acc}(1)$, and let

  $$M = (\mathsf{Acc}(1)\ /\ (C/E).E'\ /\ A) \qquad \text{and} \qquad M' = (C\ /\ E\ /\ A)$$

  Then we have

  $$M \mapsto M'$$

and

$$\begin{array}{rcl} \overline{M} & = & \bullet\,\{\Diamond\}\left[\,\overline{(C/E)}\cdot\overline{E'}\,\right]\{\overline{A}\} \\ & \xrightarrow{K} & \overline{(C/E)}\,\{\overline{A}\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is $\mathsf{Acc}(n+1)$, and let

$$M = (\mathsf{Acc}(n+1)\ /\ V.E\ /\ A) \qquad \text{and} \qquad M' = (\mathsf{Acc}(n)\ /\ E\ /\ A)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{rcl} \overline{M} & = & \underline{n}\,[\uparrow]\,[\,\overline{V}\cdot\overline{E}\,]\,\{\overline{A}\} \\ & \xrightarrow{K} & \underline{n}\,[\,\uparrow\circ(\overline{V}\cdot\overline{E})\,]\,\{\overline{A}\} \\ & \xrightarrow{K} & \underline{n}\,[\,\overline{E}\,]\,\{\overline{A}\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is Grab.

$$M = (\mathsf{Grab};\ C\ /\ E\ /\ V.A) \qquad \text{and} \qquad M' = (C\ /\ V.E\ /\ A)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{rcl} \overline{M} & = & (\lambda\,\langle\!\langle C\rangle\!\rangle)\,[\,\overline{E}\,]\,\{\overline{V}\cdot\overline{A}\} \\ & \xrightarrow{K} & \langle\!\langle C\rangle\!\rangle\,[\,\overline{V}\cdot\overline{E}\,]\,\{\overline{A}\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is Push.

$$(\mathsf{Push}(C);\ C'\ /\ E\ /\ A) \qquad \text{and} \qquad (C'\ /\ E\ /\ (C/E).A)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{rcl}
\overline{M} & = & \langle\!\langle C' \rangle\!\rangle \, \{\langle\!\langle C \rangle\!\rangle \cdot \Diamond\} \, [\,\overline{E}\,] \, \{\overline{A}\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{(\langle\!\langle C \rangle\!\rangle \cdot \Diamond) \, [\,\overline{E}\,]\} \, \{\overline{A}\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{(\langle\!\langle C \rangle\!\rangle \cdot \Diamond) \, [\,\overline{E}\,] \mathbin{@} \overline{A}\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{\big(\langle\!\langle C \rangle\!\rangle \, [\,\overline{E}\,] \cdot \Diamond \, [\,\overline{E}\,]\big) \mathbin{@} \overline{A}\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{\big(\langle\!\langle C \rangle\!\rangle \, [\,\overline{E}\,] \cdot \Diamond\big) \mathbin{@} \overline{A}\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{\langle\!\langle C \rangle\!\rangle \, [\,\overline{E}\,] \cdot \big(\Diamond \mathbin{@} \overline{A}\big)\} \\
 & \xrightarrow{K} & \langle\!\langle C' \rangle\!\rangle \, [\,\overline{E}\,] \, \{\langle\!\langle C \rangle\!\rangle \, [\,\overline{E}\,] \cdot \overline{A}\} \\
 & = & \overline{M'}
\end{array}$$

Since the K-strategy is deterministic and cannot progress when the corresponding Krivine machine state cannot progress, the simulation shown above also supports the following proposition:

Proposition 4.4.3. *If* $\overline{M} \xrightarrow{K}{}^{*} t$*, then there exists a Krivine state* N *such that* $M \mapsto^{*} N$ *and* $t \xrightarrow{K}{}^{*} \overline{N}$*.*

Chapter 5

Call-by-value Machines

We have thus demonstrated an excellent correspondence between the $\overline{\lambda}\sigma_w$-calculus and a call-by-name abstract machine. Is it possible to accomplish a similar task with a call-by-value machine? The answer is *yes*, but we need an additional feature in the $\overline{\lambda}\sigma_w$-calculus in order to maintain the nice properties of the call-by-name case.

5.1 The $\tilde{\mu}$ Operator

The reformulation of the $\overline{\lambda}$-calculus by Curien and Herbelin in [4] introduced the use of two control operators: μ and $\tilde{\mu}$. The μ operator was borrowed directly from Parigot [17] and was, for Curien and Herbelin, a key element of constructing terms representing proofs in classical sequent calculus (just as Parigot had used it for classical natural deduction). The $\tilde{\mu}$ control operator is the dual of the μ operator and allows an encoding of λ-terms into the $\overline{\lambda}$-calculus that permits call-by-value reduction (left-to-right or right-to-left) to take place at the top of the term.

The $\overline{\lambda}\sigma_w$-calculus does not have an explicit μ control operator because $\overline{\lambda}\sigma_w$-terms correspond to proofs in the sequent calculus for minimal propositional logic instead of classical logic. The use of the μ operator for controlling the substitution of contexts for context variables is not necessary, since there can be only one context variable in scope at any given time (corresponding to $\Diamond$ in the $\overline{\lambda}\sigma_w$-calculus).

Figure 5.1: Syntax of the $\overline{\lambda}\sigma_w$-calculus

$$\boxed{\begin{array}{rcl} t & ::= & \bullet\{e\} \mid (\lambda t) \mid t\{e\} \mid t[s] \\ e & ::= & \Diamond \mid (t \cdot e) \mid (e \mathbin{@} e') \mid (\tilde{\mu} t) \mid e[s] \\ s & ::= & \mathtt{id} \mid (t \cdot s) \mid (s \circ s') \mid \uparrow \end{array}}$$

However, the $\tilde{\mu}$ control operator does have an important function, even in the case of a calculus corresponding to minimal logic: it allows a simulation of call-by-value abstract machines, while maintaining the property that all reduction steps can be performed in the top of the syntax tree. So we present a version of the $\overline{\lambda}\sigma_w$-calculus with $\tilde{\mu}$ (calling it the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus). The $\tilde{\mu}$ provides one new syntactic case for contexts (Figure 5.1), and one new typing rule (Figure 5.2). There are two new reduction rules to accommodate the $\tilde{\mu}$ operator and these are shown in Figure 5.3. With these changes, we are ready to define translations for the SECD and ZINC abstract machines.

5.2 SECD Abstract Machine

We will examine a version of the SECD machine that has been modified somewhat from Plotkin's presentation [18]. The original machine operated on terms that required no compilation, but performed an on-the-fly compilation step of exactly the same nature as the one observed in the addition machine in Chapter 3: changing an operation from infix to postfix. In this case the operation is application instead of addition. We have removed this step from the machine and added a compilation function to perform this structural change in advance.

We have also added an accumulator, *i.e.* a position for a single value, in order to delay the process of putting a value on the stack. A piece of code that would have been put directly onto the stack in the original machine will now go to the accumulator first. Then it may immediately be put on top of the stack with a Push instruction, or it may be applied to the arguments on top of the stack with an Apply instruction. Thus, with respect to the original machine, the accumulator simply represents what would have been the top item on the stack.

Figure 5.2: Assignment of types to $\overline{\lambda}\tilde{\mu}\sigma_w$-terms

$$\frac{}{\Gamma \mid .:A \;\vdash\; .\{\Diamond\} : A} \qquad \frac{\Gamma, A \mid .:A \;\vdash\; .\{e\} : B}{\Gamma, A \;\vdash\; \bullet\{e\} : B}$$

$$\frac{\Gamma \;\vdash\; t : A \qquad \Gamma \mid .:B \;\vdash\; .\{e\} : C}{\Gamma \mid .:A \to B \;\vdash\; .\{t \cdot e\} : C} \qquad \frac{\Gamma, A \;\vdash\; t : B}{\Gamma \;\vdash\; \lambda t : A \to B}$$

$$\frac{\Gamma \mid .:C \;\vdash\; .\{e\} : A \qquad \Gamma \mid .:A \;\vdash\; .\{e'\} : B}{\Gamma \mid .:C \;\vdash\; .\{e \mathbin{@} e'\} : B}$$

$$\frac{\Gamma \;\vdash\; t : A \qquad \Gamma \mid .:A \;\vdash\; .\{e\} : B}{\Gamma \;\vdash\; t\{e\} : B}$$

$$\frac{\Gamma, A \;\vdash\; t : B}{\Gamma \mid .:A \;\vdash\; .(\tilde{\mu}t) : B}$$

$$\frac{s : (\Gamma \Rightarrow \Delta) \qquad \Gamma \mid .:A \;\vdash\; .\{e\} : B}{\Delta \mid .:A \;\vdash\; .\{e[s]\} : B} \qquad \frac{s : (\Gamma \Rightarrow \Delta) \qquad \Gamma \;\vdash\; t : A}{\Delta \;\vdash\; t[s] : A}$$

$$\frac{}{\mathtt{id} : (\Gamma \Rightarrow \Gamma)}$$

$$\frac{}{\uparrow : (\Gamma \Rightarrow \Gamma, A)}$$

$$\frac{s : (\Gamma \Rightarrow \Delta) \qquad \Delta \;\vdash\; t : A}{t \cdot s : (\Gamma, A \Rightarrow \Delta)}$$

$$\frac{s : (\Gamma \Rightarrow \Delta) \qquad s' : (\Delta \Rightarrow \Sigma)}{s \circ s' : (\Gamma \Rightarrow \Sigma)}$$

Figure 5.3: Reduction rules of the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus

(BetaNil)	$(\lambda t)\,[\,s\,]\,\{\lozenge\}$	$\rightarrow$	$(\lambda t)\,[\,s\,]$
(BetaCons)	$(\lambda t)\,[\,s\,]\,\{t' \cdot e\}$	$\rightarrow$	$t\,[\,t' \cdot s\,]\,\{e\}$
(MuTilde)	$t\,\{(\tilde{\mu}t')\,[\,s\,]\}$	$\rightarrow$	$t'\,[\,t \cdot s\,]$
(MuTildeConcat)	$t\,\{(\tilde{\mu}t')\,[\,s\,]\,@\,e\}$	$\rightarrow$	$t'\,[\,t \cdot s\,]\,\{e\}$
(AppApp)	$t\,\{e\}\,\{e'\}$	$\rightarrow$	$t\,\{e\,@\,e'\}$
(ConcatNil)	$\lozenge\,@\,e$	$\rightarrow$	e
(ConcatCons)	$(t \cdot e)\,@\,e'$	$\rightarrow$	$t \cdot (e\,@\,e')$
(ConcatAssoc)	$(e\,@\,e')\,@\,e''$	$\rightarrow$	$e\,@\,(e'\,@\,e'')$
(SubVarNil)	$\bullet\,\{\lozenge\}\,[\,t \cdot s\,]$	$\rightarrow$	t
(SubVarCons)	$\bullet\,\{e\}\,[\,t \cdot s\,]$	$\rightarrow$	$t\,\{e\,[\,t \cdot s\,]\}$
(SubApp)	$t\,\{e\}\,[\,s\,]$	$\rightarrow$	$t\,[\,s\,]\,\{e\,[\,s\,]\}$
(SubSubTerm)	$t\,[\,s\,]\,[\,s'\,]$	$\rightarrow$	$t\,[\,s \circ s'\,]$
(SubNil)	$\lozenge\,[\,s\,]$	$\rightarrow$	$\lozenge$
(SubCons)	$(t \cdot e)\,[\,s\,]$	$\rightarrow$	$t\,[\,s\,] \cdot e\,[\,s\,]$
(SubSubContext)	$e\,[\,s\,]\,[\,s'\,]$	$\rightarrow$	$e\,[\,s \circ s'\,]$
(SubConcat)	$(e\,@\,e')\,[\,s\,]$	$\rightarrow$	$e\,[\,s\,]\,@\,e'\,[\,s\,]$
(CompNil)	$\mathtt{id} \circ s$	$\rightarrow$	s
(CompCons)	$(t \cdot s) \circ s'$	$\rightarrow$	$t[s'] \cdot (s \circ s')$
(CompShift)	$\uparrow \circ\,(t \cdot s)$	$\rightarrow$	s
(CompAssoc)	$(s \circ s') \circ s''$	$\rightarrow$	$s \circ (s' \circ s'')$

These changes have a two-fold purpose. First, they allow a simpler presentation of the decompilation to the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus. Second, these changes highlight the underlying similarity between the SECD and the ZINC machines.

5.2.1 Definition

The machine code comprises five instructions, divided here into two syntactic categories:

$$\begin{array}{rcl} C_t & ::= & \mathsf{Acc}(n);\ C_e \\ & | & \mathsf{Closure}(C_t);\ C_e \\ C_e & ::= & \mathsf{Push};\ C_t \\ & | & \mathsf{Apply};\ C_e \\ & | & \mathsf{Return} \end{array}$$

Using the syntactic categories is useful when working with an accumulator. The instructions in the first category (C_t) are operations that expect no value in the accumulator. The instructions in the second category (C_e) operate on the value in the accumulator. The transition rules are designed so that the accumulator always has a value when it should. Such an approach also eliminates a class of stuck states. The compilation function will respect this syntactic structure for the code.

The other machine components include the machine values (V), environments (E), the working stack (S), and the return stack (or *dump*) (D). The definitions of these are given in Figure 5.4 and should be reminiscent of those for the Krivine machine. However, instead of the Krivine machine's single stack, we now have a stack for values and one for continuations. These must be distinct in a call-by-value machine, but they can be one and the same in the call-by-name case.

The transitions of the SECD machine are given in Figure 5.5. The set of final states are those in which the next instruction is Return, but the return stack is empty. The other two cases in which the machine cannot progress are when the instruction is $\mathsf{Acc}(n)$, but the environment does not have n values, or when the instruction is Apply, but the working stack is empty. Both of these can be shown to be impossible if compiling closed terms with the compilation function that will be presented.

We must note that the SECD machine does not take steps of strictly bounded size. The reason is that the $\mathsf{Acc}(n)$ instruction is executed in a single step for any value n. This, in turn, is a result of the fact that the structure of the SECD machine does not naturally allow destructive use of the current environment, as was possible in the Krivine machine. We could add a new component to the SECD machine state for holding a temporary environment that could be walked through one item at a time if we felt it were necessary to do so. However, in practice, it is often possible to execute this step in constant time by means of pointer arithmetic; so the level of abstractness of this machine is somewhat a matter of perspective, and we will work with the machine as it is.

Figure 5.4: Components of the SECD abstract machine

(Machine Values)	V	$::=$	(C_t/E)
(Environment Stacks)	E	$::=$	$\varepsilon \mid V.E$
(Working Stacks)	S	$::=$	$\varepsilon \mid V.S$
(Return Stacks)	D	$::=$	$\varepsilon \mid (C_e, S, E) :: D$
(Machine States)	M	$::=$	$(- \;/\; C_t \;/\; S \;/\; E \;/\; D)$
		$\mid$	$(V \;/\; C_e \;/\; S \;/\; E \;/\; D)$

Figure 5.5: Transition rules for the SECD abstract machine

$$\begin{array}{c}
(- \;/\; \mathsf{Acc}(n);\; C_e \;/\; S \;/\; V_0..V_n.E \;/\; D) \mapsto \\
\qquad (V_n \;/\; C_e \;/\; S \;/\; V_0..V_n.E \;/\; D) \\
(- \;/\; \mathsf{Closure}(C_t);\; C_e \;/\; S \;/\; E \;/\; D) \mapsto \\
\qquad ((C_t/E) \;/\; C_e \;/\; S \;/\; E \;/\; D) \\
(V \;/\; \mathsf{Push};\; C_e \;/\; S \;/\; E \;/\; D) \mapsto \\
\qquad (- \;/\; C_t \;/\; V.S \;/\; E \;/\; D) \\
((C_t/E) \;/\; \mathsf{Apply};\; C_e \;/\; V.S \;/\; E' \;/\; D) \mapsto \\
\qquad (- \;/\; C_t \;/\; \varepsilon \;/\; V.E \;/\; (C_e, S, E') :: D) \\
(V \;/\; \mathsf{Return} \;/\; S' \;/\; E' \;/\; (C_e, S, E) :: D) \mapsto \\
\qquad (V \;/\; C_e \;/\; S \;/\; E \;/\; D)
\end{array}$$

5.2.2 Compiling and Loading

As for the Krivine machine, we will use λ_{DB}-terms as the source language for simplicity. Compiling λ_{DB}-terms to SECD machine code is done as follows:

$$\begin{array}{rcl} [\![\underline{n}]\!] & = & \mathsf{Acc}(n) \\ [\![\lambda t]\!] & = & \mathsf{Closure}([\![t]\!];\ \mathsf{Return}) \\ [\![(t\ u)]\!] & = & [\![u]\!];\ \mathsf{Push};\ [\![t]\!];\ \mathsf{Apply} \end{array}$$

Abstractions are compiled into a Closure instruction. In the SECD machine, we will be dealing with proper call-by-value closures, which pair an environment with an abstraction. The function $\mathcal{L}$ is the compile-and-load function:

$$\mathcal{L}(t) \quad = \quad (-\ /\ [\![t]\!];\ \mathsf{Return}\ /\ \varepsilon\ /\ \varepsilon\ /\ \varepsilon)$$

The extra Return instruction is necessary, as it is the only way to end a block of code according to the grammar. Our use of Return is identical to the use of an empty code sequence in the original machine.

5.2.3 Decompilation to the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus

We now present the decompilation from SECD machine states to $\overline{\lambda}\tilde{\mu}\sigma_w$-terms. The idea behind this translation is that we will use the top of the current environment (and explicit substitution in the calculus) to store the values on the working stack. So, the decompilation will be parameterized by an integer p, which will represent the size of the working stack. Unfortunately, this means that the translation is not purely compositional, as was the one for the Krivine machine. Decompilation of code is as follows:

$$\begin{array}{rcl} \langle\!\langle \mathsf{Acc}(n);\ C_e \rangle\!\rangle_p & = & \underline{n+p}\ \left\{ \langle\!\langle C_e \rangle\!\rangle_p \right\} \\ \langle\!\langle \mathsf{Closure}(C_t);\ C_e \rangle\!\rangle_p & = & (\lambda \langle\!\langle C_t \rangle\!\rangle_0)\,[\uparrow]^p \left\{ \langle\!\langle C_e \rangle\!\rangle_p \right\} \\ \\ \langle\!\langle \mathsf{Push};\ C_t \rangle\!\rangle_p & = & \tilde{\mu} \langle\!\langle C_t \rangle\!\rangle_{p+1} \\ \langle\!\langle \mathsf{Apply};\ C_e \rangle\!\rangle_p & = & (\bullet\,\{\Diamond\}) \cdot \langle\!\langle C_e \rangle\!\rangle_{p-1}\,[\uparrow] \\ \langle\!\langle \mathsf{Return} \rangle\!\rangle_p & = & \Diamond \end{array}$$

The rest of the components of the machine can be decompiled without a parameter. Here is the decompilation of machine values (*i.e.* closures):

$$\overline{(C_t/E)} \quad = \quad (\lambda \langle\!\langle C_t \rangle\!\rangle_0)\,[\,\overline{E}\,]$$

Next we define the translation for sequences of values (such as the working stack or the environment) and also a notation for describing their size:

$$\begin{array}{rcl} \overline{\varepsilon} & = & \mathtt{id} \\ \overline{V.S} & = & \overline{V} \cdot \overline{S} \\ \\ |\varepsilon| & = & 0 \\ |V.S| & = & 1 + |S| \end{array}$$

We will use the notation S_1, S_2 for representing the concatenation of these two sequences of values.

The return stack has the following decompilation:

$$\begin{array}{rcl} \overline{\varepsilon} & = & \Diamond \\ \overline{(C_e, S, E) :: D} & = & \langle\!\langle C_e \rangle\!\rangle_{|S|} \left[\overline{S, E} \right] @ \overline{D} \end{array}$$

And the decompilation of the machine states puts these components together:

$$\begin{array}{rcl} \overline{(- \ / \ C_t \ / \ S \ / \ E \ / \ D)} & = & \langle\!\langle C_t \rangle\!\rangle_{|S|} \left[\overline{S, E} \right] \{ \overline{D} \} \\ \overline{(V \ / \ C_e \ / \ S \ / \ E \ / \ D)} & = & \overline{V} \left\{ \langle\!\langle C_e \rangle\!\rangle_{|S|} \left[\overline{S, E} \right] @ \overline{D} \right\} \end{array}$$

5.2.4 The SECD Strategy in the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus

Our correspondence between the SECD machine and the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus is unfortunately not quite as direct as was that for the Krivine machine. The first issue is that, after the machine reaches a final state, the corresponding $\overline{\lambda}\tilde{\mu}\sigma_w$-term will be able to make one additional reduction step. This could actually be remedied by the addition a new type of machine state, to which the current final machine states would progress. Then we would translate this new machine state to the corresponding final $\overline{\lambda}\tilde{\mu}\sigma_w$-term. However, instead of doing this, we will simply restate the proposition about final states.

The second issue is that our method of managing the working stack prevents us from strictly bounding the number of $\overline{\lambda}\tilde{\mu}\sigma_w$-reduction steps that will be taken for any given machine state. However, we *can* put a static bound on this number because we can statically determine that maximum value of p, the parameter used to decompile code. In light of these considerations, here is this proposition about the translation:

Proposition 5.2.1. *There exists a deterministic strategy S in the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus such that*

a. the reduction steps performed by S are confined to a fixed distance from the top of the syntax tree

b. if M is a final state in the Krivine machine, then $\overline{M}$ reduces in one step to a S-value

Figure 5.6: The U strategy

$$e\,[\,\uparrow \circ (t \cdot s)\,] \xrightarrow{U} e\,[\,s\,]$$

$$e\,[\,s\,]\,[\,s'\,] \xrightarrow{U} e\,[\,s \circ s'\,]$$

Figure 5.7: The V strategy

$$t\,[\,\uparrow \circ (t' \cdot s)\,] \xrightarrow{V} t\,[\,s\,]$$

$$t\,[\,s\,]\,[\,s'\,] \xrightarrow{V} t\,[\,s \circ s'\,] \qquad t\,\{e\}\,[\,s\,] \xrightarrow{V} t\,[\,s\,]\,\{e\,[\,s\,]\}$$

$$\bullet\,\{\Diamond\}\,[\,t \cdot s\,] \xrightarrow{V} t$$

c. if $M \mapsto M'$ in the SECD machine, then $\overline{M} \xrightarrow{S}{}^{n} \overline{M'}$, for some value n that is dependent upon the machine's original starting state

As was done for the K strategy, we decompose the strategy S into sub-strategies to make the presentation easier, but the strategy still specifies reductions only in a fixed part of the term tree. The sub-strategies must have a priority assigned to the rules that overlap, and similarly, we present the rules so that those rules that are visually higher on the page have precedence over the ones below them.

The V strategy is unchanged. We now have a U strategy (Figure 5.6), which performs a similar function on contexts. The Q strategy (Figure 5.8) is a modified version of the P strategy in the last chapter. The strategy J' (Figure 5.9) is identical to the previous J strategy, except that is makes use of the Q instead of P. Finally, the S strategy is shown in Figure 5.10. In comparison with the K strategy, it changes the priorities of the rules and adds a rule for

Figure 5.8: The Q strategy

$$\Diamond\,[\,s\,] \xrightarrow{Q} \Diamond \qquad (t \cdot e)\,[\,s\,] \xrightarrow{Q} t\,[\,s\,] \cdot e\,[\,s\,]$$

$$\frac{e \xrightarrow{U} e'}{t \cdot e \xrightarrow{Q} t \cdot e'}$$

$$\frac{t \xrightarrow{V} t'}{t \cdot e \xrightarrow{Q} t' \cdot e}$$

Figure 5.9: The J' strategy

$$\frac{e \xrightarrow{Q} e'}{e \,@\, e'' \xrightarrow{J'} e' \,@\, e''}$$

$$\Diamond \,@\, e \xrightarrow{J'} e \qquad (t \cdot e) \,@\, e' \xrightarrow{J'} t \cdot (e \,@\, e') \qquad t \cdot (\Diamond \,@\, e) \xrightarrow{J'} t \cdot e$$

Figure 5.10: The S strategy

$$t\,\{e\}\,\{e'\} \xrightarrow{S} t\,\{e \,@\, e'\}$$

$$\frac{t \xrightarrow{V} t'}{t\,\{e\} \xrightarrow{S} t'\,\{e\}}$$

$$(\lambda t)\,[\,s\,]\,\{t' \cdot e\} \xrightarrow{S} t\,[\,t' \cdot s\,]\,\{e\} \qquad t'\,\{(\tilde{\mu} t)\,[\,s\,] \,@\, e\} \xrightarrow{S} t\,[\,t' \cdot s\,]\,\{e\}$$

$$\frac{e \xrightarrow{J'} e'}{t\,\{e\} \xrightarrow{S} t\,\{e'\}}$$

the $\tilde{\mu}$ operator.

5.2.5 Simulation

To confirm Proposition 5.2.1, we will check the translation of final states. If

$$M = (V \;/\; \mathsf{Return} \;/\; S \;/\; E \;/\; \varepsilon)$$

then

$$\overline{M} = \overline{V}\,\{\Diamond \;@\; \Diamond\}$$

which reduces to

$$\overline{V}\,\{\Diamond\}$$

in one step. Now we may confirm the rest of the proposition by tracing the simulation of the SECD machine steps. Again, we consider cases according to the current machine instruction.

- Assume the next instruction is Acc, and let

  $$M = (- \;/\; \mathsf{Acc}(n);\; C_e \;/\; S \;/\; V_0..V_n.E \;/\; D)$$

  and

  $$M' = (V_n \;/\; C_e \;/\; S \;/\; V_0..V_n.E \;/\; D)$$

  Then we have

  $$M \mapsto M'$$

  and

  $$\begin{array}{rcl} \overline{M} & = & \underline{n+|S|}\ \left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\right\}\left[\,\overline{S, V_1..V_n.E}\,\right]\{\overline{D}\} \\ & \xrightarrow{S} & \underline{n+|S|}\ \left[\,\overline{S, V_1..V_n.E}\,\right]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\left[\,\overline{S, V_1..V_n.E}\,\right]\right\}\{\overline{D}\} \\ & \xrightarrow{S} & \underline{n+|S|}\ \left[\,\overline{S, V_1..V_n.E}\,\right]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\left[\,\overline{S, V_1..V_n.E}\,\right] @\, \overline{D}\right\} \\ & \xrightarrow{S} & \overline{V_n}\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\left[\,\overline{S, V_1..V_n.E}\,\right] @\, \overline{D}\right\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is $\mathsf{Closure}$, and let

  $$(- \;/\; \mathsf{Closure}(C_t);\; C_e \;/\; S \;/\; E \;/\; D)$$

  and

  $$((C_t/E) \;/\; C_e \;/\; S \;/\; E \;/\; D)$$

  Then we have

  $$M \mapsto M'$$

and

$$\begin{array}{rcl}
\overline{M} & = & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\uparrow]^{|S|}\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\right\}[\,\overline{S,E}\,]\,\{\overline{D}\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\uparrow]^{|S|}\,[\,\overline{S,E}\,]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\right\}\{\overline{D}\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\uparrow]^{|S|}\,[\,\overline{S,E}\,]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\;@\;\overline{D}\right\} \\
& \xrightarrow{S}{}^{*} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\;@\;\overline{D}\right\} \\
& = & \overline{M'}
\end{array}$$

- Assume the next instruction is Push, and let

$$M = (V \;/\; \mathsf{Push};\; C_e \;/\; S \;/\; E \;/\; D)$$

and

$$M' = (- \;/\; C_t \;/\; V.S \;/\; E \;/\; D)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{rcl}
\overline{M} & = & \overline{V}\left(\left(\tilde{\mu}\langle\!\langle C_t\rangle\!\rangle_{|S|+1}\right)[\,\overline{S,E}\,]\;@\;\overline{D}\right) \\
& \xrightarrow{S} & \langle\!\langle C_t\rangle\!\rangle_{|S|+1}\,[\,\overline{V}\cdot\overline{S,E}\,]\,\{\overline{D}\} \\
& = & \overline{M'}
\end{array}$$

- Assume the next instruction is Apply, and let

$$M = ((C_t/E) \;/\; \mathsf{Apply};\; C_e \;/\; V.S \;/\; E' \;/\; D)$$

and

$$M' = (- \;/\; C_t \;/\; \varepsilon \;/\; V.E \;/\; (C_e, S, E') :: D)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{lll}
\overline{M} & = & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\Big((\bullet\{\Diamond\})\cdot\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\uparrow]\Big)\,[\,\overline{V}\cdot\overline{S,E}\,]\ @\ \overline{D}\right\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\Big((\bullet\{\Diamond\})\,[\,\overline{V}\cdot\overline{S,E}\,]\cdot\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\uparrow]\,[\,\overline{V}\cdot\overline{S,E}\,]\Big)\ @\ \overline{D}\right\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\Big((\bullet\{\Diamond\})\,[\,\overline{V}\cdot\overline{S,E}\,]\cdot\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\uparrow\circ\overline{V}\cdot\overline{S,E}\,]\Big)\ @\ \overline{D}\right\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\Big((\bullet\{\Diamond\})\,[\,\overline{V}\cdot\overline{S,E}\,]\cdot\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\Big)\ @\ \overline{D}\right\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\Big(\overline{V}\cdot\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\Big)\ @\ \overline{D}\right\} \\
& \xrightarrow{S} & (\lambda\langle\!\langle C_t\rangle\!\rangle_0)\,[\,\overline{E}\,]\left\{\overline{V}\cdot\Big(\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\ @\ \overline{D}\Big)\right\} \\
& \xrightarrow{S} & \langle\!\langle C_t\rangle\!\rangle_0\,[\,\overline{V}\cdot\overline{E}\,]\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\ @\ \overline{D}\right\} \\
& = & \overline{M'}
\end{array}$$

- Assume the next instruction is Return, and let

$$M = (V\ /\ \mathsf{Return}\ /\ S'\ /\ E'\ /\ (C_e, S, E) :: D)$$

and

$$M' = (V\ /\ C_e\ /\ S\ /\ E\ /\ D)$$

Then we have

$$M \mapsto M'$$

and

$$\begin{array}{lll}
\overline{M} & = & \overline{V}\left\{\Diamond\,[\,\overline{S',E'}\,]\ @\ \Big(\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\ @\ \overline{D}\Big)\right\} \\
& \xrightarrow{S} & \overline{V}\left\{\Diamond\ @\ \Big(\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\ @\ \overline{D}\Big)\right\} \\
& \xrightarrow{S} & \overline{V}\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}\,[\,\overline{S,E}\,]\ @\ \overline{D}\right\} \\
& = & \overline{M'}
\end{array}$$

In several cases, the number of steps taken is dependent upon the size of the working stack, S. A maximum value for this can be statically determined. As in the case of the Krivine machine, this simulation supports the complimentary one that has to be qualified to take into account the issue with the final states.

Proposition 5.2.2. *If $\overline{M} \xrightarrow{S}{}^{*} t$, then there exists an SECD machine state N such that, either*

- *N is a final state and $\overline{N} \xrightarrow{S} t$, or*
- *$M \mapsto^{*} N$ and $t \xrightarrow{S}{}^{*} \overline{N}$.*

5.3 ZINC Abstract Machine

Leroy [15] described the ZINC abstract machine as “Krivine’s machine with marks specialized to call-by-value only, and extended to handle constants as well.” We will be ignoring the features of the ZINC machine that handle constants, and instead work with a portion that manipulates closures as its only values. However, after examining the modified version of the SECD machine in the last section, it is perhaps easier to describe the ZINC machine as a version of the SECD machine that has been extended for handling multiple applications. This extension also allows an optimization on the environments.

The ZINC machine’s use of multiple application is one of its distinguishing features, and we will see how this can be nicely translated to the argument lists derived from the $\overline{\lambda}$-calculus. Unfortunately, the optimization on environments does not have special significance in the calculus and will, in fact, complicate both the compilation and decompilation. In the ZINC machine, there is a *near environment* and a *far environment*. The near environment keeps the bindings of the arguments to formal parameters that result from the most recent (multiple) application. When necessary, these are moved to the far environment. There is a machine instruction for accessing the near environment (Acc) and one for accessing the far environment (EnvAcc). During the decompilation to $\overline{\lambda}\tilde{\mu}\sigma_w$-terms, the distinction between the near environment and far environment will be lost.

5.3.1 Definition

The machine instructions are essentially a superset of those of the SECD machine:

$$\begin{array}{rcl}
C_t & ::= & \mathsf{Acc}(n);\ C_e \\
 & | & \mathsf{EnvAcc}(n);\ C_e \\
 & | & \mathsf{Closure}(C_t);\ C_e \\
 & | & \mathsf{Grab};\ C_t \\
C_e & ::= & \mathsf{Push};\ C_t \\
 & | & \mathsf{Apply}(n);\ C_e \\
 & | & \mathsf{Return}
\end{array}$$

The instruction Apply now takes a parameter. $\mathsf{Apply}(1)$ corresponds to the Apply instruction of the SECD machine.[1]

[1]The original machine presented by Leroy in [15] had an Apply instruction that took no argument. It simply applied a function to all the arguments that were pushed since the previous $\mathsf{Pushmark}$ instruction. The actual implementation of the ZINC machine as the bytecode interpreter of Objective Caml uses both forms of Apply. We chose to work with the $\mathsf{Apply}(n)$ form because it makes our decompilation much simpler. Similarly, the actual machine uses a Return instruction that takes an argument, and we could have simplified our decompilation

Figure 5.11: Components of the ZINC abstract machine

$$\begin{array}{llcl}
\text{(Machine Values)} & V, W & ::= & (C_t/E/F) \\
\text{(Environment Stacks)} & E, F & ::= & \varepsilon \mid V.E \\
\text{(Working Stacks)} & S & ::= & \varepsilon \mid V.S \\
\text{(Argument Stacks)} & A & ::= & \varepsilon \mid V.S \\
\text{(Return Stacks)} & D & ::= & \varepsilon \mid (C_e, S, E, F, A) :: D \\
\text{(Machine States)} & M & ::= & (- \;/\; C_t \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D) \\
 & & \mid & (V \;/\; C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D)
\end{array}$$

The components of the ZINC machine are similar to those of the SECD machine and are shown in Figure 5.11. In general, we will use E for the current near environment and F for the current far environment. The closures will now have to keep both parts of the environment. The argument stack (A) is new with respect to the simpler SECD machine. Since a function can be applied to more arguments than it takes, the argument stack acts as a buffer to hold the arguments that have already been "applied" but not yet "grabbed." It has a strong relationship to the stack used by the Krivine machine.

The transition rules for the ZINC machine are given in Figure 5.12. The final states are almost the same as those of the SECD machine: A state is final if there is a Return instruction but the return stack is empty *and* the argument stack is empty. In this machine there are now two types of steps whose size cannot be strictly bounded: the environment access steps (as before) and the application step, which operates upon multiple arguments.

5.3.2 Compiling and Loading

The compilation of λ_{DB}-terms is more difficult now that we must manage two environments. Thus, it is parameterized by the length of the near environment q.

$$\begin{array}{rcl}
[\![\underline{n}]\!]_q & = & \mathsf{Acc}(n) \quad \text{if } n < q \\
[\![\underline{n}]\!]_q & = & \mathsf{EnvAcc}(n-q) \quad \text{if } n \geq q \\
[\![\lambda^n t]\!]_q & = & \mathsf{Closure}(\mathsf{Grab}^{n-1};\ [\![t]\!]_n;\ \mathsf{Return}) \\
[\![(t\ u_1\ \ldots u_n)]\!]_q & = & [\![u_n]\!]_q;\ \mathsf{Push}; \ldots;\ [\![u_1]\!]_q;\ \mathsf{Push};\ [\![t]\!]_q;\ \mathsf{Apply}(n)
\end{array}$$

The compile-and-load function is similar to the previous ones.

$$\mathcal{L}(t) \quad = \quad (- \;/\; [\![t]\!];\ \mathsf{Return} \;/\; \varepsilon \;/\; \varepsilon \;/\; \varepsilon \;/\; \varepsilon \;/\; \varepsilon)$$

even further by using this. However, we decided to leave the Return alone to highlight the similarity between the ZINC and SECD machines.

Figure 5.12: Transition rules for the ZINC abstract machine

$$
\begin{array}{l}
(- \;/\; \mathsf{Acc}(n);\ C_e \;/\; S \;/\; V_1..V_n.E \;/\; F \;/\; A \;/\; D) \mapsto \\
\qquad\qquad (V_n \;/\; C_e \;/\; S \;/\; V_1..V_n.E \;/\; F \;/\; A \;/\; D) \\
(- \;/\; \mathsf{EnvAcc}(n);\ C_e \;/\; S \;/\; E \;/\; W_1..W_n.F \;/\; A \;/\; D) \mapsto \\
\qquad\qquad (W_n \;/\; C_e \;/\; S \;/\; E \;/\; W_1..W_n.F \;/\; A \;/\; D) \\
(- \;/\; \mathsf{Closure}(C_t);\ C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D) \mapsto \\
\qquad\qquad ((C_t/-/E,F) \;/\; C_e \;/\; T \;/\; \varepsilon \;/\; E,F \;/\; A \;/\; D) \\
(- \;/\; \mathsf{Grab};\ C_t \;/\; \varepsilon \;/\; E' \;/\; F' \;/\; \varepsilon \;/\; (C_e,S,E,F,A) :: D) \mapsto \\
\qquad\qquad ((C_t/E'/F') \;/\; C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D) \\
(- \;/\; \mathsf{Grab};\ C_t \;/\; \varepsilon \;/\; E \;/\; F \;/\; V.A \;/\; D) \mapsto \\
\qquad\qquad (- \;/\; C_t \;/\; \varepsilon \;/\; V.E \;/\; F \;/\; A \;/\; D) \\
(V \;/\; \mathsf{Push};\ C_t \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D) \mapsto \\
\qquad\qquad (- \;/\; C_t \;/\; V.S \;/\; E \;/\; F \;/\; A \;/\; D) \\
((C_t/E'/F') \;/\; \mathsf{Apply}(n);\ C_e \;/\; V_1..V_n.S \;/\; E \;/\; F \;/\; A \;/\; D) \mapsto \\
\qquad\qquad (- \;/\; C_t \;/\; \varepsilon \;/\; V_1.E' \;/\; F' \;/\; V_2..V_n \;/\; (C_e,S,E,F,A) :: D) \\
(V \;/\; \mathsf{Return} \;/\; S' \;/\; E' \;/\; F' \;/\; \varepsilon \;/\; (C_e,S,E,F,A) :: D) \mapsto \\
\qquad\qquad (V \;/\; C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D) \\
((C_t/E'/F') \;/\; \mathsf{Return} \;/\; S \;/\; E \;/\; F \;/\; V.A \;/\; D) \mapsto \\
\qquad\qquad (- \;/\; C_t \;/\; \varepsilon \;/\; V.E' \;/\; F' \;/\; A \;/\; D)
\end{array}
$$

5.3.3 Decompilation to the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus

The decompilation follows that of the SECD machine, except that now we must add an extra parameter to keep track of the size of the near environment so that we can properly decompile the EnvAcc instructions. Notice the use of the argument lists in the decompilation of the Apply instruction.

$$
\begin{array}{rcl}
\langle\!\langle \mathsf{Acc}(n);\ C_e \rangle\!\rangle_p^q & = & \underline{n+p}\ \left\{ \langle\!\langle C_e \rangle\!\rangle_p^q \right\} \\
\langle\!\langle \mathsf{EnvAcc}(n);\ C_e \rangle\!\rangle_p^q & = & \underline{n+p+q}\ \left\{ \langle\!\langle C_e \rangle\!\rangle_p^q \right\} \\
\langle\!\langle \mathsf{Closure}(C_t);\ C_e \rangle\!\rangle_p^q & = & \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^1 \right) [\, \uparrow^p] \left\{ \langle\!\langle C_e \rangle\!\rangle_p^0 \right\} \\
\langle\!\langle \mathsf{Grab};\ C_t \rangle\!\rangle_p^q & = & \lambda \langle\!\langle C_t \rangle\!\rangle_p^{q+1} \\
\\
\langle\!\langle \mathsf{Push};\ C_t \rangle\!\rangle_p^q & = & \tilde{\mu} \langle\!\langle C_t \rangle\!\rangle_{p+1}^q \\
\langle\!\langle \mathsf{Apply}(n);\ C_e \rangle\!\rangle_p^q & = & (\underline{1} \cdots \underline{n} \cdot \Diamond) \;@\; \langle\!\langle C_e \rangle\!\rangle_{p-n}^q [\, \uparrow^n] \\
\langle\!\langle \mathsf{Return} \rangle\!\rangle_p^q & = & \Diamond
\end{array}
$$

Decompilation of closures joins the two parts of the environment together:

$$
\overline{(C_t/E/F)} \quad = \quad \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^{|E|+1} \right) [\, \overline{E,F}\,]
$$

There is no change in the decompilation of sequences of values:

$$\begin{array}{rcl} \overline{\varepsilon} & = & \texttt{id} \\ \overline{V.S} & = & \overline{V} \cdot \overline{S} \end{array}$$

Decompilation of the extra argument stack is the same as the decompilation of the Krivine machine stack:

$$\begin{array}{rcl} \overline{\varepsilon} & = & \Diamond \\ \overline{V.A} & = & \overline{V} \cdot \overline{A} \end{array}$$

The return stack must now account for the argument stack component:

$$\begin{array}{rcl} \overline{\varepsilon} & = & \Diamond \\ \overline{(C_e, S, E, F, A) :: D} & = & \langle\!\langle C_e \rangle\!\rangle_{|S|}^{|E|} \left[\, \overline{S, E, F} \,\right] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D}\right) \end{array}$$

Finally, machine states are decompiled as follows:

$$\begin{array}{rcl} \overline{(- \;/\; C_t \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D)} & = & \langle\!\langle C_t \rangle\!\rangle_{|S|}^{|E|} \left[\, \overline{S, E, F} \,\right] \left\{\overline{A} \mathbin{@} \overline{D}\right\} \\ \overline{(V \;/\; C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D)} & = & \overline{V} \left\{ \langle\!\langle C_e \rangle\!\rangle_{|S|}^{|E|} \left[\, \overline{S, E, F} \,\right] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D}\right) \right\} \end{array}$$

5.3.4 The ZINC Strategy in the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus

When trying to describe the correspondence between the machine and calculus, we encounter the same issues that arise in the SECD machine. However, there is a more significant issue, too. Because of the need to propagate substitutions through a list of arguments in the step corresponding to the Apply instruction, we can no longer claim that our strategy will only make changes to a fixed part of the term. Thus, our strategy can no longer be a machine with steps of strictly-bounded size. There must be one recursive rule added to the strategy to propagate substitutions. Here is our proposition for the ZINC machine:

Proposition 5.3.1. *There exists a deterministic strategy Z in the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus such that*

a. if M is a final state in the Krivine machine, then $\overline{M}$ reduces in two steps to a Z-value

b. if $M \mapsto M'$ in the ZINC machine, then $\overline{M} \xrightarrow{Z}{}^{n} \overline{M'}$, for some value n that is dependent upon the machine's original starting state

The components of the strategy are presented in Figures 5.13, 5.14, 5.15, 5.16, and 5.17. The recursively defined part of the transition relation is the R strategy in Figure 5.15.

Figure 5.13: The U strategy

$$e\,[\,\uparrow \circ (t \cdot s)\,] \xrightarrow{U} e\,[\,s\,]$$

$$e\,[\,s\,]\,[\,s'\,] \xrightarrow{U} e\,[\,s \circ s'\,]$$

Figure 5.14: The V strategy

$$t\,[\,\uparrow \circ (t' \cdot s)\,] \xrightarrow{V} t\,[\,s\,]$$

$$t\,[\,s\,]\,[\,s'\,] \xrightarrow{V} t\,[\,s \circ s'\,] \qquad t\,\{e\}\,[\,s\,] \xrightarrow{V} t\,[\,s\,]\,\{e\,[\,s\,]\}$$

$$\bullet\,\{\Diamond\}\,[\,t \cdot s\,] \xrightarrow{V} t$$

Figure 5.15: The R strategy

$$\Diamond\,[\,s\,] \xrightarrow{R} \Diamond \qquad (t \cdot e)\,[\,s\,] \xrightarrow{R} t\,[\,s\,] \cdot e\,[\,s\,]$$

$$\frac{t \xrightarrow{V} t'}{t \cdot e \xrightarrow{R} t' \cdot e}$$

$$\frac{e \xrightarrow{R} e'}{t \cdot e \xrightarrow{R} t \cdot e'}$$

Figure 5.16: The Y strategy

$$(e \mathbin{@} e')\,[\,s\,] \mathbin{@} e'' \xrightarrow{Y} (e\,[\,s\,] \mathbin{@} e'\,[\,s\,]) \mathbin{@} e'' \qquad (e \mathbin{@} e') \mathbin{@} e'' \xrightarrow{Y} e \mathbin{@} (e' \mathbin{@} e'')$$

$$\frac{e_2 \xrightarrow{U} e_2'}{e_1 \mathbin{@} (e_2 \mathbin{@} e_3) \xrightarrow{Y} e_1 \mathbin{@} (e_2' \mathbin{@} e_3)}$$

$$\frac{e \xrightarrow{R} e'}{e \mathbin{@} e'' \xrightarrow{Y} e' \mathbin{@} e''}$$

$$\Diamond \mathbin{@} e \xrightarrow{Y} e \qquad (t \cdot e) \mathbin{@} e' \xrightarrow{Y} t \cdot (e \mathbin{@} e') \qquad t \cdot (\Diamond \mathbin{@} e) \xrightarrow{Y} t \cdot e$$

Figure 5.17: The Z strategy

$$t\,\{e\}\,\{e'\} \xrightarrow{Z} t\,\{e \mathbin{@} e'\}$$

$$\frac{t \xrightarrow{V} t'}{t\,\{e\} \xrightarrow{Z} t'\,\{e\}}$$

$$(\lambda t)\,[\,s\,]\,\{t' \cdot e\} \xrightarrow{Z} t\,[\,t' \cdot s\,]\,\{e\} \qquad t'\,\{(\tilde{\mu} t)\,[\,s\,] \mathbin{@} e\} \xrightarrow{Z} t\,[\,t' \cdot s\,]\,\{e\}$$

$$\frac{e \xrightarrow{Y} e'}{t\,\{e\} \xrightarrow{Z} t\,\{e'\}}$$

5.3.5 Simulation

The first part of Proposition 5.3.1 concerns the final states. We can check that if

$$M = (V \;/\; \mathsf{Return} \;/\; S \;/\; E \;/\; F \;/\; A \;/\; \varepsilon)$$

then

$$\overline{M} = \overline{V}\,\{\Diamond \mathbin{@} (\Diamond \mathbin{@} \Diamond)\}$$

and

$$\overline{M} \xrightarrow{Z}{}^{2}\, \overline{V}\,\{\Diamond\}$$

The rest of the proposition can be confirmed by considering each possible machine step.

- Assume the next instruction is Acc, and let

  $$M = (- \;/\; \mathsf{Acc}(n);\; C_e \;/\; S \;/\; V_1..V_n.E \;/\; F \;/\; A \;/\; D)$$

  and

  $$M' = (V_n \;/\; C_e \;/\; S \;/\; V_1..V_n.E \;/\; F \;/\; A \;/\; D)$$

  Then

  $$M \mapsto M'$$

  and

  $$\begin{aligned}
  \overline{M} \quad &= \quad \underline{n+|S|}\ \left\{\langle\!\langle C_e\rangle\!\rangle^{n+|E|}_{|S|}\right\}\,[\,\overline{S, V_1..V_n.E, F}\,]\,\{\overline{A} \mathbin{@} \overline{D}\}\\
  &\xrightarrow{Z} \quad \underline{n+|S|}\ [\,\overline{S, V_1..V_n.E, F}\,]\,\left\{\langle\!\langle C_e\rangle\!\rangle^{n+|E|}_{|S|}\,[\,\overline{S, V_1..V_n.E, F}\,]\right\}\{\overline{A} \mathbin{@} \overline{D}\}\\
  &\xrightarrow{Z} \quad \underline{n+|S|}\ [\,\overline{S, V_1..V_n.E, F}\,]\,\left\{\langle\!\langle C_e\rangle\!\rangle^{n+|E|}_{|S|}\,[\,\overline{S, V_1..V_n.E, F}\,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D}\right)\right\}\\
  &\xrightarrow{Z}{}^{*} \quad \overline{V_n}\left\{\langle\!\langle C_e\rangle\!\rangle^{n+|E|}_{|S|}\,[\,\overline{S, V_1..V_n.E, F}\,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D}\right)\right\}\\
  &= \quad \overline{M'}
  \end{aligned}$$

- Assume the next instruction is EnvAcc, and let

  $$M = (- \ / \ \mathsf{EnvAcc}(n); \ C_e \ / \ S \ / \ E \ / \ W_1..W_n.F \ / \ A \ / \ D)$$

  and

  $$M' = (W_n \ / \ C_e \ / \ S \ / \ E \ / \ W_1..W_n.F \ / \ A \ / \ D)$$

  $$\begin{array}{rl}
\overline{M} \quad = & \underline{n+|S|+|E|} \ \left\{ \langle\!\langle C_e \rangle\!\rangle^{|E|}_{|S|} \right\} [\, \overline{S,E,W_1..W_n.F} \,] \, \{ \overline{A} \mathbin{@} \overline{D} \} \\
\xrightarrow{Z} & \underline{n+|S|+|E|} \ [\, \overline{S,E,W_1..W_n.F} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{|E|}_{|S|} \, [\, \overline{S,E,W_1..W_n.F} \,] \right\} \{ \overline{A} \mathbin{@} \overline{D} \} \\
\xrightarrow{Z} & \underline{n+|S|+|E|} \ [\, \overline{S,E,W_1..W_n.F} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{|E|}_{|S|} \, [\, \overline{S,E,W_1..W_n.F} \,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D} \right) \right\} \\
\xrightarrow{Z}{}^{*} & \overline{W_n} \left\{ \langle\!\langle C_e \rangle\!\rangle^{|E|}_{|S|} \, [\, \overline{S,E,W_1..W_n.F} \,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D} \right) \right\} \\
= & \overline{M'}
\end{array}$$

- Assume the next instruction is $\mathsf{Closure}$, and let

  $$M = (- \ / \ \mathsf{Closure}(C_t); \ C_e \ / \ S \ / \ E \ / \ F \ / \ A \ / \ D)$$

  and

  $$M' = ((C_t/-/E,F) \ / \ C_e \ / \ T \ / \ \varepsilon \ / \ E,F \ / \ A \ / \ D)$$

  Then

  $$M \mapsto M'$$

  and

  $$\begin{array}{rl}
\overline{M} \quad = & \left(\lambda \langle\!\langle C_t \rangle\!\rangle^{1}_{0} \right) [\uparrow]^{|S|} \left\{ \langle\!\langle C_e \rangle\!\rangle^{0}_{|S|} \right\} [\, \overline{S,E,F} \,] \, \{ \overline{A} \mathbin{@} \overline{D} \} \\
\xrightarrow{Z} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle^{1}_{0} \right) [\uparrow]^{|S|} \, [\, \overline{S,E,F} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{0}_{|S|} \, [\, \overline{S,E,F} \,] \right\} \{ \overline{A} \mathbin{@} \overline{D} \} \\
\xrightarrow{Z} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle^{1}_{0} \right) [\uparrow]^{|S|} \, [\, \overline{S,E,F} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{0}_{|S|} \, [\, \overline{S,E,F} \,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D} \right) \right\} \\
\xrightarrow{Z}{}^{*} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle^{1}_{0} \right) [\, \overline{E,F} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{0}_{|S|} \, [\, \overline{S,E,F} \,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D} \right) \right\} \\
= & \overline{M'}
\end{array}$$

- Assume the next instruction is Grab and no arguments are left in the argument stack. If we let

  $$M = (- \ / \ \mathsf{Grab}; \ C_t \ / \ \varepsilon \ / \ E' \ / \ F' \ / \ \varepsilon \ / \ (C_e, S, E, F, A) :: D)$$

  and

  $$M' = ((C_t/E'/F') \ / \ C_e \ / \ S \ / \ E \ / \ F \ / \ A \ / \ D)$$

  Then

  $$M \mapsto M'$$

  and

  $$\begin{array}{rl}
\overline{M} \quad = & \left(\lambda \langle\!\langle C_t \rangle\!\rangle^{|E'|+1}_{0} \right) [\, \overline{E',F'} \,] \left\{ \langle\!\langle C_e \rangle\!\rangle^{|E|}_{|S|} \, [\, \overline{S,E,F} \,] \mathbin{@} \left(\overline{A} \mathbin{@} \overline{D} \right) \right\} \\
= & \overline{M'}
\end{array}$$

- Assume the next instruction is Grab, and the argument stack is not empty. If we let
$$M = (- \;/\; \mathsf{Grab};\; C_t \;/\; \varepsilon \;/\; E \;/\; F \;/\; V.A \;/\; D)$$
and
$$M' = (- \;/\; C_t \;/\; \varepsilon \;/\; V.E \;/\; F \;/\; A \;/\; D)$$
Then
$$M \mapsto M'$$
and
$$\begin{array}{rcl} \overline{M} & = & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E|+1}\right)\left[\,\overline{E,F}\,\right]\left\{\left(\overline{V}\cdot\overline{A}\right) @ \overline{D}\right\} \\ & \xrightarrow{Z} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E|+1}\right)\left[\,\overline{E,F}\,\right]\left\{\overline{V}\cdot\left(\overline{A} @ \overline{D}\right)\right\} \\ & \xrightarrow{Z} & \langle\!\langle C_t\rangle\!\rangle_0^{|E|+1}\left[\,\overline{V}\cdot\overline{E,F}\,\right]\left\{\overline{A} @ \overline{D}\right\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is Push, and let
$$M = (V \;/\; \mathsf{Push};\; C_t \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D)$$
and
$$M' = (- \;/\; C_t \;/\; V.S \;/\; E \;/\; F \;/\; A \;/\; D)$$
Then
$$M \mapsto M'$$
and
$$\begin{array}{rcl} \overline{M} & = & \overline{V}\left\{\left(\tilde{\mu}\langle\!\langle C_t\rangle\!\rangle_{|S|+1}^{|E|}\right)\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right\} \\ & \xrightarrow{Z} & \langle\!\langle C_t\rangle\!\rangle_{|S|+1}^{|E|}\left[\,\overline{V}\cdot\overline{S,E,F}\,\right]\left\{\overline{A} @ \overline{D}\right\} \\ & = & \overline{M'} \end{array}$$

- Assume the next instruction is Apply, and let
$$M = ((C_t/E'/F') \;/\; \mathsf{Apply}(n);\; C_e \;/\; V_1..V_n.S \;/\; E \;/\; F \;/\; A \;/\; D)$$
and
$$M' = (- \;/\; C_t \;/\; \varepsilon \;/\; V_1.E' \;/\; F' \;/\; V_2..V_n \;/\; (C_e, S, E, F, A) :: D)$$
Then
$$M \mapsto M'$$

and

$$\begin{array}{rcl}
\overline{M} & = & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{\left((\underline{1}\cdots\underline{n}\cdot\Diamond) @ \langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\,[\uparrow]^n\right)\left[\,\overline{V_1..V_n.S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right\}} \\
& \xrightarrow{Z} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{\left((\underline{1}\cdots\underline{n}\cdot\Diamond)\left[\,\overline{V_1..V_n.S,E,F}\,\right] @ \langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\,[\uparrow]^n\left[\,\overline{V_1..V_n.S,E,F}\,\right]\right) @ \left(\overline{A} @ \overline{D}\right)\right\}} \\
& \xrightarrow{Z} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{(\underline{1}\cdots\underline{n}\cdot\Diamond)\left[\,\overline{V_1..V_n.S,E,F}\,\right] @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\,[\uparrow]^n\left[\,\overline{V_1..V_n.S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\}} \\
& \xrightarrow{Z} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{(\underline{1}\cdots\underline{n}\cdot\Diamond)\left[\,\overline{V_1..V_n.S,E,F}\,\right] @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\}} \\
& \xrightarrow{Z}{}^{*} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{(\overline{V_1}\cdots\overline{V_n}\cdot\Diamond) @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\}} \\
& \xrightarrow{Z} & \left(\lambda\langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\right)\left[\,\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{\overline{V_1}\cdot\left((\overline{V_2}\cdots\overline{V_n}\cdot\Diamond) @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right)\right\}} \\
& \xrightarrow{Z} & \langle\!\langle C_t\rangle\!\rangle_0^{|E'|+1}\left[\,\overline{V_1}\cdot\overline{E',F'}\,\right] \\
\multicolumn{3}{l}{\left\{(\overline{V_2}\cdots\overline{V_n}\cdot\Diamond) @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\}} \\
& = & \overline{M'}
\end{array}$$

- Assume the next instruction is Return and no arguments are left on the argument stack. If we let

  $$M = (V \;/\; \mathsf{Return} \;/\; S' \;/\; E' \;/\; F' \;/\; \varepsilon \;/\; (C_e, S, E, F, A) :: D)$$

  and

  $$M' = (V \;/\; C_e \;/\; S \;/\; E \;/\; F \;/\; A \;/\; D)$$

  Then

  $$M \mapsto M'$$

  and

  $$\begin{array}{rcl}
\overline{M} & = & \overline{V}\left\{\Diamond\left[\,\overline{S',E',F'}\,\right] @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\} \\
& \xrightarrow{Z} & \overline{V}\left\{\Diamond @ \left(\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right)\right\} \\
& \xrightarrow{Z} & \overline{V}\left\{\langle\!\langle C_e\rangle\!\rangle_{|S|}^{|E|}\left[\,\overline{S,E,F}\,\right] @ \left(\overline{A} @ \overline{D}\right)\right\} \\
& = & \overline{M'}
\end{array}$$

- Assume the next instruction is Return, and the argument stack is not empty. If we let

$$M = ((C_t/E'/F') \;/\; \mathsf{Return} \;/\; S \;/\; E \;/\; F \;/\; V.A \;/\; D)$$

and

$$M' = (- \;/\; C_t \;/\; \varepsilon \;/\; V.E' \;/\; F' \;/\; A \;/\; D)$$

Then

$$M \mapsto M'$$

and

$$\begin{array}{rcl} \overline{M} & = & \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^{|E'|+1}\right) \left[\,\overline{E', F'}\,\right] \left\{\Diamond \left[\,\overline{S, E, F}\,\right] @ \left(\left(\overline{V} \cdot \overline{A}\right) @ \overline{D}\right)\right\} \\ & \xrightarrow{Z} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^{|E'|+1}\right) \left[\,\overline{E', F'}\,\right] \left\{\Diamond @ \left(\left(\overline{V} \cdot \overline{A}\right) @ \overline{D}\right)\right\} \\ & \xrightarrow{Z} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^{|E'|+1}\right) \left[\,\overline{E', F'}\,\right] \left\{\left(\overline{V} \cdot \overline{A}\right) @ \overline{D}\right\} \\ & \xrightarrow{Z} & \left(\lambda \langle\!\langle C_t \rangle\!\rangle_0^{|E'|+1}\right) \left[\,\overline{E', F'}\,\right] \left\{\overline{V} \cdot \left(\overline{A} @ \overline{D}\right)\right\} \\ & \xrightarrow{Z} & \langle\!\langle C_t \rangle\!\rangle_0^{|E'|+1} \left[\,\overline{V} \cdot \overline{E', F'}\,\right] \left\{\overline{A} @ \overline{D}\right\} \\ & = & \overline{M'} \end{array}$$

Finally, we can make the corresponding complimentary proposition:

Proposition 5.3.2. *If $\overline{M} \xrightarrow{Z}{}^{*} t$, then there exists an ZINC machine state N such that, either*

- *N is a final state and $\overline{N} \xrightarrow{Z}{}^{2} t$, or*
- *$M \mapsto^{*} N$ and $t \xrightarrow{Z}{}^{*} \overline{N}$.*

Conclusion

This thesis has demonstrated that the level of detail found in abstract machines designed for weak β-reduction can be simulated within a term reduction system based upon sequent calculus cut elimination. The idea that a calculus can function at the level of an abstract machine is incipient in the work of Curien and Herbelin [4] but could not be fully realized without a version of the $\overline{\lambda}$-calculus supporting simultaneous explicit substitutions and closures for weak reduction. We believe that that $\overline{\lambda}\sigma_w$- and the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculi demonstrate the accessibility of this goal.

It is typical to think of abstract machines as a means by which one can break down large tasks into small steps; however, the simulations in this thesis demonstrate how each machine step is further broken down into smaller steps in the $\overline{\lambda}\sigma_w$-calculus. If we see the deterministic strategy for the rewriting of $\overline{\lambda}\sigma_w$-terms as an abstract machine (which it literally is), then the operation of the Krivine, SECD, and ZINC abstract machines can be seen as the result of optimizing a lower-level machine by joining smaller instructions into larger macro-instructions. Our translations show that using these macro-instructions often does not result in more than a constant-time speed increase over the execution of steps according to the smaller instructions.

We believe that all weak reduction strategies for the λ-calculus can be encoded into strategies with constant-sized reduction steps in the $\overline{\lambda}\tilde{\mu}\sigma_w$-calculus. However, strong reduction is an entirely different issue—one that we have not considered. There may be a version of the $\overline{\lambda}$-calculus analogous to the $\lambda\sigma_{\Uparrow}$-calculus, but it is unclear what could be said about the step sizes of strong reduction strategies in such a calculus.

Another direction for future research would be to use the type system of the $\overline{\lambda}$-calculus to assign types to machine code or to machine states. Potentially, a type system based on LJT could be ideal for typing low-level code. This proposal is supported by the fact that some machine instructions in the call-by-value machines were naturally translated into $\overline{\lambda}$-terms while others were naturally translated into $\overline{\lambda}$-contexts. Additionally, we have yet to investigate the classical sequent calculus, which would likely offer a useful correspondence with machines that implement control operators.

In this work, we have thus demonstrated that an important connection exists between the LJT sequent calculus and some abstract machines. The research of Herbelin [10], Curien and Herbelin [4], and Hardin, Maranget, and Pagano

[9] has been the central foundation for this. We believe that our research has elucidated some important new points and has demonstrated that this is a fruitful direction for further study.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[2] H. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. 1992.

[3] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[4] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 233–243. ACM Press, 2000.

[5] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: linear logic. In J.-Y. Girard, Y. Lafont, and L. Régnier, editors, *Proceedings of the Workshop on Linear Logic*. Cornell, 1993.

[6] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[7] G. Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[8] Timothy G. Griffin. The formulae-as-types notion of control. In *Conference Record of the 17th Annual ACM Symposium on the Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990*, pages 47–57, New York, 1990. ACM Press.

[9] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional back-ends within the lambda-sigma calculus. Technical Report RR-3034, INRIA, November 1996.

[10] Hugo Herbelin. A lambda-calculus structure isomorphic to sequent calculus structure, draft. Available from the author's homepage: `http://coq.inria.fr/~herbelin/LAMBDA-BAR-FULL.dvi.gz`, 1994.

[11] Hugo Herbelin. Explicit substitutions and reducibility. *Journal of Logic and Computation*, 11(3):429–449, 2001.

[12] W. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[13] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

[14] Jean-Louis Krivine. A call-by-name lambda calculus machine. To appear in Higher Order and Symbolic Computation, 2004.

[15] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[16] Pierre Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 60–69, New York, NY, 1994.

[17] M. Parigot. Classical proofs as programs. *Computational Logic and Theory*, 713:263–276, 1993.

[18] G. D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[19] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[20] D. Prawitz. *Natural Deduction, a Proof-Theoretical Study.* Almquist and Wiksell, Stockholm, 1965.

[21] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, 2003.

[22] Mitchell Wand. On the correctness of the krivine machine. Submitted for publication, October 2003.