Sequent Calculi and Abstract Machines

ZENA M. ARIOLA
University of Oregon
AARON BOHANNON
University of Pennsylvania
and
AMR SABRY
Indiana University

We propose a sequent calculus derived from the $\overline{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin that is expressive enough to directly represent the fine details of program evaluation using typical abstract machines. Not only does the calculus easily encode the usual components of abstract machines such as environments and stacks, but it can also simulate the transition steps of the abstract machine with just a constant overhead. Technically this is achieved by ensuring that reduction in the calculus always happens at a bounded depth from the root of the term. We illustrate these properties by providing *shallow encodings* of the Krivine (call-by-name) and the CEK (call-by-value) abstract machines in the calculus.

Categories and Subject Descriptors: F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational Semantics*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms: Languages,Theory

Additional Key Words and Phrases: Curry-Howard Isomorphism, Duality, Explicit Substitutions, Krivine Machine, Natural Deduction

1. 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 [Howard 1980]. The core of this relationship is a correspondence between formal proofs in a logical inference system and terms of a programming language. The most basic instance of this connection is the one between minimal natural deduction and the λ-calculus. According to this paradigm, propositions correspond to types and proofs to programs: a program t of type τ is also seen as a proof of proposition τ. The reduction rules of the λ-calculus correspond to proof normalization steps. Griffin [1990] extended this isomorphism to classical natural deduction and an extension of the λ-calculus with

Supported by National Science Foundation grant number CCR-0204389.

control operators. Curien and Herbelin [2000] further extended the correspondence to sequent calculi. They show that different ways of executing a program can be observed at the level of the logic.

Based on the pioneering work of Plotkin [1975], programming calculi are now routinely used to model the semantics of programming languages. The fundamental theorem that makes a calculus suitable to describe a *programming language* is the *standardization theorem*. The semantics of a programming language is ultimately described with an evaluator, which is a partial function mapping programs to answers. The standardization theorem mediates between the axioms of the calculus and the evaluator by giving a deterministic way to apply the axioms that leads to an answer, if one exists.

When modeled in the calculus, the deterministic strategy of the evaluator typically consists of a loop which repeatedly *searches* for the next redex and then applies one of the axioms. In most known cases (*e.g.* the leftmost-outermost strategy), the search phase might go arbitrarily deep in the syntax tree looking for a redex. This however is not satisfactory if one wants to capture the execution of an abstract machine, which always executes a redex at a fixed position. This implies the need to introduce a new criterion to characterize a calculus as a "good calculus" for reasoning about machines. In other words, we need an analogue to the *standardization theorem* to mediate the relationship between a calculus and a machine. The criterion we present in this paper consists of the following:

> The reductions corresponding to the abstract machine transitions must occur at a *bounded depth* from the root of the syntax tree.

Although languages based on lambda-calculi have been successful in supporting reasoning about high-level programs, in this paper we focus on sequent calculi. We show that sequent calculi are more suitable for reasoning about abstract machines since they allow one to define an evaluator for terms in a *tail-recursive* fashion, thus satisfying the above criterion. A tail-recursive evaluator captures the dispatch function carried out by a machine. In contrast, an evaluator for terms corresponding to natural deduction proofs is not tail recursive, requiring an unbounded search for the next redex.

We present a sequent calculus which naturally embeds run-time data-structures, such as control stacks and environments. We show how such a calculus allows one to simulate the executions of two abstract machines: the Krivine machine and the CEK machine. The Krivine machine corresponds to a deterministic call-by-name reduction strategy. Whereas, the CEK corresponds to a call-by-value reduction strategy. Both strategies are defined in a *non-recursive*, *i.e.*, without using recursion, manner. Moreover, they are obtained by choosing a different orientation of a critical pair of the reduction semantics.

Our approach of capturing the operational semantics of the machine directly through the normalization of the calculus is in contrast to specifying the operational semantics on top of the calculus. More precisely, our approach entails the following:

—The state of the abstract machine is captured as a *term*.

—The transitions of the abstract machine are captured as *term reductions*.

Our approach provides a *shallow embedding* of an abstract machine in a calculus/logic, as opposed to a *deep embedding*. This allows reasoning about the machine *inside* the logic itself instead of on *top* of it. It also allows an elegant and simple formulation of safety, based on the subject reduction theorem of the calculus.

1.1 Outline of the paper

Section 2 discusses related work. Sections 3 and 4 are background material. Section 3 introduces minimal and classical natural deduction and their computational interpretation. λ-calculus and the λ_{DB}-calculus are presented as term assignments for minimal natural deduction proofs. λ-calculus corresponds to the system with the collection of assumptions organized as a set. The λ_{DB}-calculus instead of variables to name assumptions has de Bruijn indices, it corresponds to the system with the assumptions maintained in a sequence. The extension of λ-calculus with control operators is presented as a term assignment for classical logic. Section 4 introduces Gentzen's sequent calculus (minimal LJ and classical LK) and a variant called $\mathrm{LK}_{\mu\tilde{\mu}}$. Curien and Herbelin $\overline{\lambda}\mu\tilde{\mu}$-calculus [2000] is introduced as a term assignment for this variant.

Section 5 compares λ-calculus and the $\overline{\lambda}\mu\tilde{\mu}$-calculus. One key and distinctive feature of the $\overline{\lambda}\mu\tilde{\mu}$-calculus is that application occurs between a term and an *argument list*, rather than a single argument as it would in the λ-calculus in natural deduction style. The use of argument lists in the $\overline{\lambda}\mu\tilde{\mu}$-calculus grants a greater range of expressiveness in structuring terms than is available in the λ-calculus, and this more refined level of expressiveness is an essential tool for capturing the low level details of β-reduction. The $\overline{\lambda}\mu\tilde{\mu}$-calculus however is still not sufficient for simulating abstract machines, as it lacks a notion of *environment* or simultaneous substitution. Instead of extending the $\overline{\lambda}\mu\tilde{\mu}$-calculus with this notion, mirroring the $\lambda\sigma_w$-calculus [Hardin et al. 1996], we work within the $\overline{\lambda}\mu\tilde{\mu}$-calculus with unary substitution and show how to simulate the notion of an environment. The resulting calculus is called the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus.

Section 6 presents the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus, which extends the $\overline{\lambda}\mu\tilde{\mu}$-calculus with de Bruijn indices, weakening and explicit substitution. We define the notion of well-formed term and well-typed term, properties which are preserved during reduction. We present call-by-name and call-by-value reduction strategies, which as for the $\overline{\lambda}\mu\tilde{\mu}$-calculus are obtained by resolving a critical pair. Unlike the case for the λ-calculus, in the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus these semantics are specified in a *non-recursive* fashion. Section 7 defines over the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ an instruction set which is typical of abstract machines. Using these instructions, the compilation of λ_{DB} terms in $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ is presented. The compilation produces well-formed commands. Section 8 is devoted to showing how the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ calculus faithfully simulates the execution of the Krivine [2007] and CEK [Felleisen and Friedman 1986] abstract machines. It shows the correspondence between the Krivine machine and the call-by-name reduction strategy of $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus, and the correspondence between the right-to-left CEK machine and the call-by-value reduction strategy of $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$. Section 9 defines a type system for the Krivine machine and the right-to-left CEK machine, shows how the types are preserved by the translation in the $\overline{\lambda}\mu\tilde{\mu}$-calculus, and how to use types to reason about the correctness of different compilation schemes and optimizations. Section 10 concludes.

2. RELATED WORK

There has been previous work directed at modifying the λ-calculus so that β-reduction can be simulated with smaller reduction steps. An important contribution was the investigation of explicit substitutions [Abadi et al. 1990]. Lescanne [1994] offers a comparison of several versions of calculi with explicit substitutions. Curien, Hardin, and Lévy [1996] achieved an important goal 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. Hardin *et al.* [1996] propose a weak calculus of explicit substitution (the $\lambda\sigma_w$) 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. However, they fail to capture one important aspect of an abstract machine: dispatching the next instruction should not depend on the program to be executed. This in turn means that the calculus does not model important structures present in the machine.

This previously-mentioned work on calculi with explicit substitutions was, generally speaking, motivated by the goal of de-constructing 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 [1994], 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 [2000] and of Wadler [2003] in the area of proof terms for sequent calculus has elucidated some of the symmetries inherent in computation, including the duality of the call-by-name and call-by-value semantics. The two strategies are obtained by choosing a different orientation of a critical pair. Computation is described as an interaction between a consumer and a producer, by giving priority to the consumer one obtains call-by-name, whereas by giving priority to the producer one obtains call-by-value.

With the recent interest in security, formalization of abstract machines has attracted a lot of attention: encoding of the Java Virtual Machine (JVM) into Haskell [Jones 1998; Yelland 1999], ACL2 [Liu and Moore 2004], Coq [Barthe et al. 2001] and HOL/Isabelle [Klein and Strecker 2004] have been proposed. These approaches are all based on a *deep embedding* whereas we propose an approach based on a *shallow embedding* into a foundational sequent calculus/logic, in which the machine specification becomes executable. Our foundational approach does not immediately scale to the entire JVM. Although the latter machine is a stack-based abstract machine like the machines we consider in this paper, it also includes some "large" instructions that are intrinsically tied to the semantics of the Java or byte-code languages: one such instruction might for example, load and verify entire classes using a complicated data-flow analysis. Thus, to formalize the *entire* JVM using our approach would seem to require some additional *ad hoc* axioms and/or reasoning principles.

Our work is close in spirit to Foundational Proof-Carrying Code (FPCC) [Appel 2001] consisting of working with a minimal logic, the sequent logic in our case, instead of higher-order logic. In FPCC, the machine's operational semantics and the typing rules are proved as additional lemmas on top of the logic. In our approach

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

Fig. 1. Syntax of the implicational fragment of minimal propositional logic

they are expressed inside the logic itself. Higuchi and Ohori already have stressed the importance of a Curry-Howard isomorphism for low-level code [Higuchi and Ohori 2002; Ohori 2005]. They propose typing Java bytecode with an extension of intuitionistic propositional calculus. However, they do not provide a formalization of the operational semantics of the machine inside the logic itself. Also, it does not seem natural to relate bytecode to an intuitionistic logic since bytecode comes with instructions that modify the flow of control. These operators are more naturally embedded in a classical logic, since, as shown by Griffin [1990], they correspond to the double negation elimination inference rule.

3. NATURAL DEDUCTION AND λ-CALCULI

Natural deduction [Prawitz 1965] 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 is addressed shortly. Its proof system is also popular because its proofs can be read and constructed in a manner that is often considered more "natural" for humans than using Hilbert-style systems or sequent calculi.

We start with the implicational fragment of minimal propositional logic. Although simple, this logic still has an important relationship with computation. The syntax of the implicational fragment of minimal propositional logic is given in Figure 1. A formula is built from a set of atomic types (X), which we leave unspecified, and a single logical connective ($\rightarrow$), which joins two logical formulae. We use $A, B, C, \ldots$ as meta-variables that range over the set of formulae.

Natural deduction has several presentations. In Section 3.1 we present Prawitz's rules and in Section 3.2 we present natural deduction in sequent form. We also discuss different ways of managing the collection of assumptions, as sets, multisets or sequences. In Section 3.3 we discuss the impact of these different views on the structural rules. The computational interpretation of minimal natural deduction is presented in Section 3.4. If the assumptions are collected in a set then proofs of minimal natural deduction correspond to λ-calculus. If the assumptions are maintained as a sequence, then proofs correspond to the λ-calculus with de Bruijn's indices. Section 3.5 presents the extension to classical natural deduction with a brief introduction to continuations.

3.1 Prawitz' Natural Deduction

The inference rules for Prawitz' version of minimal natural deduction are given in Figure 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 means that the assumption is *active*. A closed leaf corresponds 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.

$$\frac{A \rightarrow B \quad A}{B} \rightarrow_e \qquad \frac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \rightarrow B} \rightarrow_i^x$$

Fig. 2. Prawitz' minimal natural deduction

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

Fig. 3. Minimal natural deduction in sequent style with assumptions as sets

In the introduction 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.

To know the active assumptions at any point in the proof, one needs 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.

3.2 Natural Deduction in Sequent Form

A sequent is a syntactic construct for asserting a relation between propositions—in our case between a collection of formulae and a single formula—which is written:

$$\Gamma \vdash A$$

In this example, Γ is the *antecedent* and A is the *succedent*. 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 contain an instance of an axiom in the inference system, and assumptions are internalized into the antecedents of the sequents. Therefore, to know the current collection of active assumptions, one simply looks at the left-hand side of the sequent.

When using sequents, it is necessary to specify what sort of "collection" the antecedent is. There are several possibilities: sets, multisets, sets of named formulae, and sequences are the primary candidates. We first present in Figure 3 a system where the antecedents should be interpreted as a *set* of formulae. We use the comma to indicate the union of a collection and a single formula (as in Γ, A).

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 the following observations hold:

(1) A proof tree may have associated "leaves" that are not directly connected to its branches. This arises from the fact that the axiom allows an arbitrary set of extra assumptions as in $A, B \vdash A$;

(2) Implication introduction may not occur if there are no open assumptions of the formula associated to the tree, as for example one cannot derive $B \vdash A \to B$ from $B \vdash B$;

(3) No discharge need actually take place. This arises from the fact that Γ may actually contain A, in which case $\Gamma, A = \Gamma$, as in $A \vdash A \to A$ from $A \vdash A$;

(4) All equivalent (*i.e.*, corresponding to the same formula) open assumptions in the tree must be simultaneously discharged (and hence marked with the same name) if any of them is discharged. For example, in the following proof:

$$\dfrac{[A]^x \qquad \dfrac{[A]^x}{B \to A}\to_i}{A \to (B \to A)}\to_i^x$$

both occurrences of formula A will be deleted by the second implication introduction. This is inherent in the use of sets to maintain assumptions.

Here is a simple proof in the system:

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

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

Had we decided to manage assumptions with multisets, we would only have to re-interpret the rules in Figure 3; the comma becomes the sum of multisets. 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 is:

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

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 problem is to attach names to the succedent in the 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 4. We return to this system in Section 3.4, where de Bruijn indices are discussed.

$$\boxed{\frac{}{\Gamma, A, \Gamma' \vdash A}\,Axiom \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\to_e \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\to_i}$$

Fig. 4. Minimal natural deduction in sequent style with assumptions as sequences

$$\boxed{\begin{array}{c} \dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}\,Weakening \qquad \dfrac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,Contraction \\[2ex] \dfrac{\Gamma, A, \Gamma', B, \Gamma'' \vdash C}{\Gamma, B, \Gamma', A, \Gamma'' \vdash C}\,Exchange \end{array}}$$

Fig. 5. Structural rules

$$\boxed{t \ ::= \ x \mid (\lambda x.t) \mid (t\ t')}$$

Fig. 6. Syntax of λ-calculus terms

3.3 Structural Rules

When working with multisets or sequences, it is sometimes necessary to add structural rules. The three structural rules typically considered are *Weakening*, *Contraction*, and *Exchange* (Figure 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.

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:

$$\frac{}{A \vdash A}\,Axiom$$

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

3.4 The λ-calculus and Minimal Natural Deduction

There is a natural bijection between the terms of the basic λ-calculus and proofs in minimal natural deduction. The syntax of the λ-calculus is presented in Figure 6. The method for matching proofs to λ-terms is described by the inference rules in Figure 7. In particular, whereas λ-terms correspond to proofs, formulae correspond to types. In this case, we are encoding proofs from a system that manages assumptions with sets of named formulae 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:

$$(\lambda x.t)\ u \to t\{x := u\}$$

$$\boxed{\begin{array}{c} \dfrac{}{\Gamma, x : A \ \vdash \ x : A}\ \textit{Axiom} \\[2ex] \dfrac{\Gamma \ \vdash \ t : A \rightarrow B \quad \Gamma \ \vdash \ u : A}{\Gamma \ \vdash \ (t\ u) : B}\ \rightarrow_e \qquad \dfrac{\Gamma, x : A \ \vdash \ t : B}{\Gamma \ \vdash \ (\lambda x.t) : A \rightarrow B}\ \rightarrow_i \end{array}}$$

Fig. 7. Assignment of λ-terms to proofs in minimal natural deduction

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

Fig. 8. Syntax of λ_{DB}-terms

$$\boxed{\dfrac{(|\Delta| = n)}{\Gamma, A, \Delta \ \vdash \ \underline{n+1} : A}\ \textit{Axiom} \quad \dfrac{\Gamma \ \vdash \ t : A \rightarrow B \quad \Gamma \ \vdash \ u : A}{\Gamma \ \vdash \ (t\ u) : B}\ \rightarrow_e \qquad \dfrac{\Gamma, A \ \vdash \ t : B}{\Gamma \ \vdash \ \lambda t : A \rightarrow B}\ \rightarrow_i}$$

Fig. 9. Assignment of λ_{DB}-terms to proofs in minimal natural deduction

Here we use the notation $t\{x := u\}$ as meta-syntax to describe the term t, with all free occurrences of x replaced with the term u. A term with no occurrences of a β-redex is said to be in *normal form.*

To assign terms that can represent proofs in the system with the collection of the assumptions interpreted as a sequence, one needs to slightly modify the λ-calculus. Instead of working with variables, one works with de Bruijn's indices representing the assumption's position in the sequence Γ. We call the new terms λ_{DB}-terms; their syntax is given in Figure 8. The assignment of terms to proofs and the typing rules of λ_{DB}-terms are in Figure 9. (We use the notation $|\Gamma|$ here to represent the number of assumptions in the sequence Γ.)

As mentioned by several authors [Curien et al. 1996; Lescanne 1994], we do not actually need to depend upon an externally-defined set of natural numbers; instead, we can integrate them into our calculus if we add a unary operator $\uparrow$, as described in Figure 10. There are no natural numbers here. Instead we may define them as follows:

$$\underline{1} := \bullet\uparrow \qquad \underline{2} := (\bullet\uparrow)\uparrow \qquad \underline{3} := ((\bullet\uparrow)\uparrow)\uparrow \qquad \underline{4} := (((\bullet\uparrow)\uparrow)\uparrow)\uparrow$$

The new terms correspond to a logic with an explicit weakening rule. These typing rules are given in Figure 11.

Intermezzo 1. In Section 7 the λ_{DB} terms are translated into a lower language which corresponds to a variant of natural deduction with multiple conclusions given in Figure 12.

3.5 Control Operators and Classical Natural Deduction

Classical logic is obtained by extending the set of formulae with the absurd formula $\bot$, which stands for the false proposition. The formula $\bot$ has no introduction rule and one elimination rule:

$$\frac{\Gamma, \neg A \ \vdash \ \bot}{\Gamma \vdash A}\ \textit{Reductio Ad Absurdum}$$

$$\boxed{t \ ::= \ \bullet \mid \lambda t \mid (t\ t') \mid (t \uparrow)}$$

Fig. 10. Syntax of λ_{DB}-terms with explicit weakening

$$\frac{}{\Gamma, A \ \vdash \ \bullet : A}\ \mathit{Axiom} \qquad \frac{\Gamma \ \vdash \ t : A}{\Gamma, B \ \vdash \ (t\uparrow) : A}\ \mathit{Weakening}$$

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

Fig. 11. λ_{DB}-terms and minimal natural deduction with weakening

$$\frac{}{\Gamma, A \ \vdash \ A, \Delta} \qquad \frac{\Gamma \ \vdash \ A, \Delta \quad \Gamma, A \ \vdash \ \Delta}{\Gamma \ \vdash \ \Delta}$$

$$\frac{\Gamma \ \vdash \ A \rightarrow B, \Delta}{\Gamma, A \ \vdash \ B, \Delta} \qquad \frac{\Gamma, A \ \vdash \ B, \Delta}{\Gamma \ \vdash \ A \rightarrow B, \Delta}$$

Fig. 12. Alternative deduction system with multiple conclusions

where $\neg A$ stands for the formula $A \rightarrow \bot$. A single application of the rule leads to the *Ex Falso Quodlibet* rule which says that from a contradiction any formula can be derived:

$$\frac{\Gamma \ \vdash \ \bot}{\Gamma \vdash A}\ \mathit{Ex\ Falso\ Quodlibet}$$

Griffin [1990] showed that the Curry-Howard isomorphism extends to an isomorphism between classical proofs and λ-terms with operators for manipulating first-class continuations. An example of such an operator is the operator $\mathcal{C}$ which was introduced for reasoning about Scheme programs [Felleisen et al. 1987; Felleisen et al. 1986; Felleisen and Friedman 1986]. Intuitively, an application of the $\mathcal{C}$ operator marks the current context with a label that can be later used to "jump back" and resume from that point. For example, in the term $1 + \mathcal{C}\ (\lambda k.2 + k\ 3)$ variable k gets bound to the context surrounding the $\mathcal{C}$ application which is represented as $1 + \Box$. When this context is invoked, the execution aborts the pending addition to 2 and instead re-instates the captured context with $\Box$ replaced with 3. The final answer is 4. Given this intuitive semantics, it is clear that one can represent the context $(1 + \Box)$ as a function which takes an integer to be added and never returns, *i.e.*, it is safe to give the function the type `int` $\rightarrow \bot$. Griffin adds $\mathcal{C}$ to the λ-calculus and shows that the resulting system corresponds to classical natural deduction as shown in Figure 13. As the last rule shows, the construct $\mathcal{C}(\lambda k.t)$ corresponds to the proof by contradiction inference rule.

4. SEQUENT CALCULI AND THE $\overline{\lambda}\mu\tilde{\mu}$-CALCULUS

Having considered natural deduction, we now turn to the sequent calculus, which has been appreciated, since its introduction by Gentzen [1969], for its symmetry and its applicability to automated proof search. Instead of having introduction

$$t ::= x \mid \lambda x.t \mid t\ t' \mid \mathcal{C}(\lambda k.t)$$

$$\frac{}{\Gamma, x : A \ \vdash \ x : A}\ \mathit{Axiom}$$

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

$$\frac{\Gamma, k : \neg A \ \vdash \ t : \bot}{\Gamma \ \vdash \ \mathcal{C}(\lambda k.t) : A}$$

Fig. 13. $\lambda\mathcal{C}$ and (classical) natural deduction

$$\frac{}{\Gamma, A \ \vdash \ A}\ \mathit{Axiom}$$

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

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

Fig. 14. Sequent calculus with assumptions as sets (Minimal LJ)

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.

In Section 4.1 the minimal sequent calculus LJ is presented. As for natural deduction, the issue of how to organize the collection of assumptions is discussed. In Section 4.2 we present the classical sequent calculus LK. In Section 4.3 we present the sequent calculus $\mathrm{LK}_{\mu\tilde{\mu}}$ which solves the issue of LK of having different normal (*i.e.* without cuts) proofs of the same formula. In Section 4.4 we present Curien and Herbelin $\overline{\lambda}\mu\tilde{\mu}$-calculus as a term assignment for $\mathrm{LK}_{\mu\tilde{\mu}}$.

4.1 The Minimal Subset

The inference rules of the minimal subset, called minimal LJ, are given in Figure 14, where Γ stands for a set of assumptions. 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.)

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 as a basis for computational structures.

Remark 2. In natural deduction, the presentation of the systems using sets and

$$\frac{}{\Gamma, A \vdash A}\;\textit{Axiom}$$

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

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\;\textit{Cont} \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash B}{\Gamma \vdash B}\;\textit{Cut}$$

Fig. 15. Minimal Sequent calculus with assumptions as multisets

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:

$$\dfrac{\dfrac{\dfrac{}{A \to B \vdash A \to B}\;\textit{Axiom} \quad \dfrac{}{A \to B, A \vdash A}\;\textit{Axiom}}{(A \to B) \to A, A \to B \vdash A}\;\to_l \quad \dfrac{}{(A \to B) \to A, A \to B, B \vdash B}\;\textit{Axiom}}{(A \to B) \to A, A \to B \vdash B}\;\to_l$$

This is a valid proof in the system of Figure 14 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 $\Gamma, A \to B = \Gamma$. This contraction 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 take the latter choice: the resulting system is presented in Figure 15.

4.2 The Classical Extension

Whereas classical natural deduction (with the exception of Parigot's classical natural deduction [1993]) is obtained by adding rules that break the symmetry of introduction and elimination rules, the classical sequent calculus is rendered by simply allowing multiple conclusions, as expressed in the judgment:

$$\Gamma \vdash \Delta$$

where Δ is a set of formulae. The judgment has a simple reading: the conjunction of the assumptions implies the disjunction of the conclusions. The system called LK is shown in Figure 16.

It may not be immediately clear why having a set of conclusions in the judgments makes the system classical. The intuition will be clear after we introduce the term assignments relating the logic to λ-calculi with control operations. In that term assignment, each formula in the set of conclusions will represent a "continuation." A system with one conclusion like LJ from the last section corresponds to a language with exactly one current continuation, *i.e.*, a language with no global control effects.

$$\dfrac{}{\Gamma, A \vdash A, \Delta}\ \mathit{Axiom}$$

$$\dfrac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\ \to_l \qquad \dfrac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\ \to_r$$

$$\dfrac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ \mathit{Cut}$$

Fig. 16. The LK sequent calculus

A system like LK with several conclusions corresponds to a language with several live continuation variables, *i.e.*, a language with first-class continuations.

4.3 The logic LK$_{\mu\tilde{\mu}}$

An issue with LJ or LK is that although the *Cut* rule is admissible, there are often multiple cut-free proofs of the same sequent. For example, the following two proofs of the same formula are distinct and neither contains any detour, *i.e.*, they are both in normal form. The problem, however, is that they are both associated to the same λ-term [Herbelin 1994].

Example 3.

$$\dfrac{\dfrac{\dfrac{}{A, C \vdash A}\ \mathit{Axiom} \quad \dfrac{}{A, C, B \vdash B}\ \mathit{Axiom}}{A \to B, A, C \vdash B}\ \to_l}{A \to B, A \vdash C \to B}\ \to_r$$

$$\dfrac{\dfrac{}{A \vdash A}\ \mathit{Axiom} \quad \dfrac{\dfrac{}{A, C, B \vdash B}\ \mathit{Axiom}}{A, B \vdash C \to B}\ \to_r}{A \to B, A \vdash C \to B}\ \to_l$$

A solution to this permutability of the left and right implication rules is offered by Danos [1993] and more recently by Curien and Herbelin [2000]. In the more recent solution, Curien and Herbelin define a restriction of LJ (LK) called LJ$_{\mu\tilde{\mu}}$ (LK$_{\mu\tilde{\mu}}$), which restores the bijection between cut free proofs and λ-terms in normal form. The restriction forces the application of the left-introduction rule as soon as it becomes applicable. Thus, only the first proof is a legal proof.

The logic LK$_{\mu\tilde{\mu}}$ uses three distinct judgments:

$$\Gamma \vdash A \mid \Delta \qquad \Gamma \mid A \vdash \Delta \qquad \Gamma \vdash \Delta$$

where Γ and Δ are sets of formulae. There is a difference between the formulae in Γ and Δ and the formula A explicitly mentioned in the first two judgments. The formulae in Γ or Δ are formulae that are *passive* or *unfocused.* Formula A is an *active* formula also called a *distinguished* formula. The symbol $\mid$ is used to separate the distinguished formula from the rest of the passive conclusions and passive assumptions, as shown in the first and second judgment, respectively. The presence of this extra symbol does not change the meaning of the judgments; the conjunctions of the assumptions still implies the disjunction of the conclusions.

$$\frac{}{\Gamma \mid A \vdash A, \Delta}\ \mathit{Axiom}_l \qquad \frac{}{\Gamma, A \vdash A \mid \Delta}\ \mathit{Axiom}_r$$

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

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \mid A \vdash \Delta}\ \mathit{Activate}_l \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \mid \Delta}\ \mathit{Activate}_r$$

$$\frac{\Gamma \vdash A \mid \Delta \quad \Gamma \mid A \vdash \Delta}{\Gamma \vdash \Delta}\ \mathit{Cut}$$

Fig. 17. The LK$_{\mu\tilde{\mu}}$ logic

The inference rules are shown in Figure 17. In the Axiom_r axiom, formula A is separated from the rest of the formulae in Δ which means that A is an active conclusion. Analogously, the dual axiom Axiom_l allows one to single out a specific assumption, which again becomes the active assumption, and thus is separated from the rest of the formulae in Γ with the $\mid$ symbol. The presence of active formulae is what distinguishes the implication rules and the cut rule from the corresponding LK rules. For example, notice how in the premises of the right implication rule formula B must be an active assumption. Thus, given the judgment $A \vdash A \mid B, A$, one cannot infer $\vdash A \to B \mid A$ since formula B is not active.

To focus on a specific assumption or conclusion one needs to use the *Activate* rules. These rules however only apply to judgments with no distinguished formulae and hence it should be possible to also "passivate" or unfocus the distinguished formula. This is accomplished by the *Cut* rule.

For example, given the judgment $A \vdash A \mid B, A$, to focus on B one first need to unfocus A:

$$\frac{\dfrac{}{A \vdash A \mid B, A}\ \mathit{Axiom}_r \qquad \dfrac{}{A \mid A \vdash B, A}\ \mathit{Axiom}_l}{A \vdash B, A}\ \mathit{Cut}$$

Now that the judgment has no active formulae, we focus on B using the $\mathit{Activate}_r$ rule:

$$\frac{A \vdash B, A}{A \vdash B \mid A}\ \mathit{Activate}_r$$

At this point one can apply the right introduction rule to obtain $\vdash A \to B \mid A$. Analogously, given a judgment with no active formulae either on the left or right-hand side, the $\mathit{Activate}_l$ allows one to select an assumption. For example:

$$\frac{A, A \to B, C \vdash B}{A \to B, C \mid A \vdash B} \qquad \frac{A, A \to B, C \vdash B}{A, C \mid A \to B \vdash B}$$

Example 4. The left-hand side proof of Example 3 is expressed in LK$_{\mu\tilde{\mu}}$ as

$$\boxed{\begin{array}{lll} \mathit{Terms} & v & ::= \; x \mid \lambda x.v \mid \mu\alpha.c \\ \mathit{Contexts} & e & ::= \; \alpha \mid v \cdot e \mid \tilde{\mu}x.c \\ \mathit{Commands} & c & ::= \; \langle v \mid e \rangle \end{array}}$$

Fig. 18. Syntax of the $\overline{\lambda}\mu\tilde{\mu}$-calculus

follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \to B, C \;\vdash\; A \to B \mid B} \qquad \dfrac{\dfrac{}{A \to B, A, C \;\vdash\; A \mid B} \qquad \dfrac{}{A \to B, A, C \mid B \;\vdash\; B}}{A \to B, A, C \mid A \to B \;\vdash\; B}\to_l}{A, A \to B, C \;\vdash\; B}\mathit{Cut}}{A \to B, A, C \;\vdash\; B \mid}\mathit{Activate}_r}{A \to B, A \vdash C \to B \mid}\to_r$$

As remarked in [Curien and Herbelin 2000], the lack of special rules to unfocus the distinguished formula destroys the cut elimination property. For example, the cut in the above proof cannot be eliminated.

4.4 The $\overline{\lambda}\mu\tilde{\mu}$-calculus and the logic $\mathsf{LK}_{\mu\tilde{\mu}}$

Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [2000] is a term assignment for the $\mathrm{LK}_{\mu\tilde{\mu}}$ variant of Gentzen's sequent calculus. The syntax is given in Figure 18. There are two "dual" syntactic categories: terms which are producers of values and contexts which are consumers of values. The interaction between a producer and a consumer is rendered by a command, which can also be seen as a hole filling operation. The duality of terms and contexts is also reflected at the variable level. One has two distinct sets of variables. The usual term variables range over by x, and the context variables range over by α. The context variables correspond to continuation variables.

To gain some intuition about the constructs of the calculus, we explain how to express in $\overline{\lambda}\mu\tilde{\mu}$, the λ-term $(\lambda x.\lambda y.x + y + z)\ 1\ z$. The producer corresponds to the function part of the application, that is the function $\lambda x.\lambda y.x + y + z$. The consumer corresponds to the arguments which are packaged up in a list made from the constructors "nil" (α) and "cons" ($\cdot$) as follows: $1 \cdot z \cdot \alpha$. Intuitively, the first argument 1 is waiting to consume a function; then the second argument z consumes the next function; and finally variable α indicates what to do next. Putting the producer and consumer together gives us the command $\langle \lambda x.\lambda y.x + y + z \mid 1 \cdot z \cdot \alpha \rangle$. The context $1 \cdot z \cdot \alpha$ intuitively corresponds to the context $\alpha\ ((\Box\ 1)\ z)$, and the command can be seen as filling the hole of the context with the lambda term. With the μ construct, one can give a name to a context so to invoke it later. For example, to execute the above command and return its result to the top level one writes $\mu\alpha.\langle \lambda x.\lambda y.x + y + z \mid 1 \cdot z \cdot \alpha \rangle$.

As the μ construct gives a name to a context, it is equivalent to Felleisen's $\mathcal{C}$ control operator. More precisely, $\mu\alpha.c$ can be written as $\mathcal{C}(\lambda\alpha.c)$.

Whereas μ names a context, the dual construct $\tilde{\mu}$ gives a name to a term. For example, one can read $\tilde{\mu}x.c$ as the let expression '$\mathsf{let}\ x = \Box\ \mathsf{in}\ c$'.

The type system uses the following three judgments:

$$\Gamma \;\vdash\; v : A \mid \Delta \qquad \Gamma \mid e : A \;\vdash\; \Delta \qquad c : (\Gamma \;\vdash\; \Delta)$$

$$\frac{}{\Gamma \mid \alpha : A \;\vdash\; \alpha : A, \Delta}\; Axiom_l \qquad \frac{}{\Gamma, x : A \;\vdash\; x : A \mid \Delta}\; Axiom_r$$

$$\frac{\Gamma \;\vdash\; v : A \mid \Delta \qquad \Gamma \mid e : B \;\vdash\; \Delta}{\Gamma \mid v \cdot e : A \to B \;\vdash\; \Delta} \to_l \qquad \frac{\Gamma, x : A \;\vdash\; v : B \mid \Delta}{\Gamma \;\vdash\; \lambda x.v : A \to B \mid \Delta} \to_r$$

$$\frac{c \;:\; (\Gamma, x : A \;\vdash\; \Delta)}{\Gamma \mid \tilde{\mu} x.c : A \;\vdash\; \Delta}\; Activate_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; \alpha : A, \Delta)}{\Gamma \;\vdash\; \mu\alpha.c : A \mid \Delta}\; Activate_r$$

$$\frac{\Gamma \;\vdash\; v : A \mid \Delta \qquad \Gamma \mid e : A \;\vdash\; \Delta}{\langle v \mid e \rangle \;:\; (\Gamma \;\vdash\; \Delta)}\; Cut$$

Fig. 19. Type system for the $\overline{\lambda}\mu\tilde{\mu}$-calculus

where Γ and Δ are sets of named formulae or types. Γ contains the types of the free term variables, and Δ contains the types of the free continuation variables. Whereas continuation variables in Prawitz' classical logic are kept in the left-hand side of the sequent and typed with formulae of the form $A \to \bot$, in here the continuation variables are kept in the right-hand side of the sequent and typed simply with A. The reading of the first judgment is as usual: term v has type A. The second judgment reads as: context e is waiting for something of type A. In other words, A is the type of the hole in the context. Commands are typed using the third judgment, which indicates that we have an active formula neither on the left-hand side nor on the right-hand side of the judgment.

The type system is given in Figure 19. Analogously to the $Axiom_r$, its dual axiom reads as: given that continuation variable α has type A one can safely assume that the context α is waiting for something of type A. For the $\to_l$ rule one has: given that v has type A and the list of arguments is waiting for something of type B, the newly applicative context $v \cdot e$ is waiting for a function of type $A \to B$. The reading of the *Cut* rule is: given a producer of type A and a context e waiting for something of type A, the type of the command is a judgment containing the types of the free term variables and of the free continuation variables occurring in the command. The $Activate_r$ rule turns a command into a producer. Notice that since Δ can contain multiple conclusions, one has freedom in choosing a potential output. For example, given

$$c \;:\; (\Gamma \;\vdash\; \alpha : A, \beta : B)$$

by selecting α, one obtains the producer $\mu\alpha.c$ of type A and by selecting β, one obtains the producer $\mu\beta.c$ of type B. Whereas the $Activate_r$ permits the selection of a conclusion, the $Activate_l$ (or equivalently the $\tilde{\mu}$) permits the selection of an assumption. As before, one has a choice in selecting the assumption. Thus, given

$$c \;:\; (x : B, y : A \;\vdash\; \Delta)$$

one can select x to obtain the consumer $\tilde{\mu}x.c$ of type B or select y to obtain the consumer $\tilde{\mu}y.c$ of type A.

4.5 Reduction Semantics for the $\overline{\lambda}\mu\tilde{\mu}$-calculus

The calculus has three basic reduction rules which can simulate either a call-by-value or a call-by-name semantics:

$$\begin{array}{lll} (\beta) & \langle \lambda x.v \mid v' \cdot e\rangle & \to \langle v' \mid \tilde{\mu}x.\langle v \mid e\rangle\rangle \\ (\mu) & \langle \mu\alpha.c \mid e\rangle & \to c\{\alpha := e\} \\ (\tilde{\mu}) & \langle v \mid \tilde{\mu}x.c\rangle & \to c\{x := v\} \end{array}$$

In the first reduction, a procedure is paired with a context that has an argument v' on the stack. The argument v' becomes the new producer, and the new context receives the value of v', binds it to x, and continues with the body of the procedure and the stack e. This reduction seems to model a call-by-value semantics but the issue is more subtle; it all depends on what a value is, which in turn depends on how a command $\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2\rangle$ evaluates. In the call-by-name semantics, the $\tilde{\mu}$-rule has higher priority, the argument v' is immediately absorbed (without evaluation). In the call-by-value semantics, the μ-rule has higher priority, thus forcing the evaluation of v'.

REMARK 5. The $\overline{\lambda}\mu\tilde{\mu}$ calculus is not confluent. For example, the term

$$\mu\beta.\langle \lambda x.z \mid \mu\alpha.\langle y \mid \beta\rangle \cdot \beta\rangle$$

will reduce to $\mu\beta.\langle z \mid \beta\rangle$ according to call-by-name:

$$\begin{array}{ll} \mu\beta.\langle \lambda x.z \mid \mu\alpha.\langle y \mid \beta\rangle \cdot \beta\rangle & \to_\beta \\ \mu\beta.\langle \mu\alpha.\langle y \mid \beta\rangle \mid \tilde{\mu}x.\langle z \mid \beta\rangle\rangle & \to_{\tilde{\mu}} \\ \mu\beta.\langle z \mid \beta\rangle & \end{array}$$

The same term will reduce to $\mu\beta.\langle y \mid \beta\rangle$ according to call-by-value:

$$\begin{array}{ll} \mu\beta.\langle \lambda x.z \mid \mu\alpha.\langle y \mid \beta\rangle \cdot \beta\rangle & \to_\beta \\ \mu\beta.\langle \mu\alpha.\langle y \mid \beta\rangle \mid \tilde{\mu}x.\langle z \mid \beta\rangle\rangle & \to_{\mu} \\ \mu\beta.\langle y \mid \beta\rangle & \end{array}$$

5. APPLICATIVE TERMS, ENVIRONMENTS, AND CONTEXTS

Our central claim is that sequent calculi are more appropriate for the simulation of abstract machines. The essence of this claim is that a reduction of terms corresponding to sequent calculus proofs is naturally tail-recursive. In contrast, reduction of terms corresponding to natural deduction proofs is not tail recursive, requiring an unbounded search for the next redex which may be located deep inside the term. We explain this point in Section 5.1 by comparing the structure of proofs in natural deduction and sequent calculus. There is however a problem with substitution, as discussed in Section 5.2.

5.1 The structure of applicative terms in λ and $\overline{\lambda}\mu\tilde{\mu}$ calculi

Consider the following two proofs of the same formula from the same assumptions:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash A \to B \to C \to D \quad \Gamma \vdash A}{\Gamma \vdash B \to C \to D}\to_e \quad \Gamma \vdash B}{\Gamma \vdash C \to D}\to_e \quad \Gamma \vdash C}{\Gamma \vdash D}\to_e$$

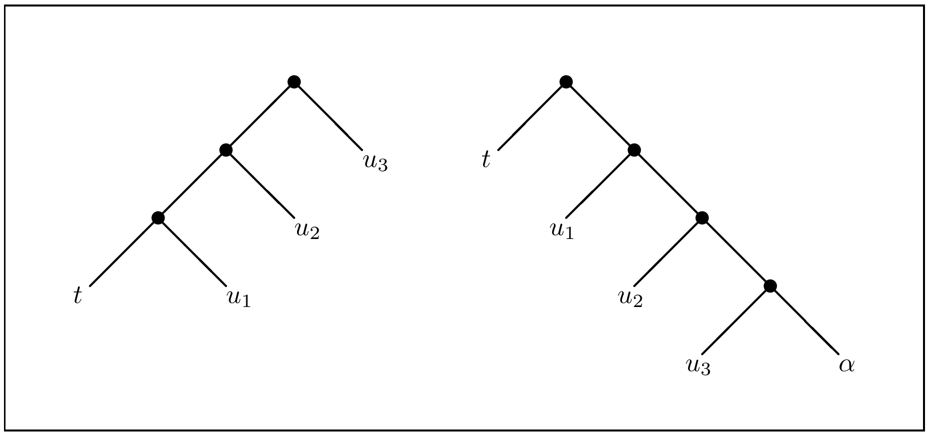

Fig. 20. The structure of applicative terms in the λ-calculus and the $\overline{\lambda}\mu\tilde{\mu}$-calculus

and

$$\dfrac{\dfrac{\Gamma \vdash A \to B \to C \to D \mid D \qquad \dfrac{\Gamma \vdash A \mid D \qquad \dfrac{\Gamma \vdash B \mid D \qquad \dfrac{\Gamma \vdash C \mid D \qquad \dfrac{}{\Gamma \mid D \vdash D}}{\Gamma \mid C \to D \vdash D}\rightarrow_l}{\Gamma \mid B \to C \to D \vdash D}\rightarrow_l}{\Gamma \mid A \to B \to C \to D \vdash D}\rightarrow_l}{\Gamma \vdash D}\,Cut}{\Gamma \vdash D \mid}\,Activate_r$$

The first proof is in the usual natural deduction style and corresponds to the typing judgment of the λ-term $(t\ u_1\ u_2\ u_3)$ as given below:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to B \to C \to D \quad \Gamma \vdash u_1 : A}{\Gamma \vdash t\ u_1 : B \to C \to D} \qquad \Gamma \vdash u_2 : B}{\Gamma \vdash t\ u_1\ u_2 : C \to D} \qquad \Gamma \vdash u_3 : C}{\Gamma \vdash t\ u_1\ u_2\ u_3 : D}$$

The second proof is in the $LK_{\mu\tilde{\mu}}$ sequent calculus and corresponds to the $\overline{\lambda}\mu\tilde{\mu}$-command $\langle t \mid u_1{\cdot}u_2{\cdot}u_3{\cdot}\alpha\rangle$ as given below (we sometimes omit the passive conclusion D):

$$\dfrac{\dfrac{\Gamma \vdash t : A \to B \to C \to D \qquad \dfrac{\Gamma \vdash u_1 : A \qquad \dfrac{\Gamma \vdash u_2 : B \qquad \dfrac{\Gamma \vdash u_3 : C \qquad \dfrac{}{\Gamma \mid \alpha : D \vdash \alpha : D}}{\Gamma \mid u_3 \cdot \alpha : C \to D \vdash \alpha : D}}{\Gamma \mid u_2 \cdot u_3 \cdot \alpha : B \to C \to D \vdash \alpha : D}}{\Gamma \mid u_1 \cdot u_2 \cdot u_3 \cdot \alpha : A \to B \to C \to D \vdash \alpha : D}}{\langle t \mid u_1 \cdot u_2 \cdot u_3 \cdot \alpha\rangle \ :\ (\Gamma\ \vdash\ \alpha : D)}}{\Gamma \vdash \mu\alpha.\langle t \mid u_1 \cdot u_2 \cdot u_3 \cdot \alpha\rangle : D}$$

While the expressions look somewhat similar, their abstract syntax trees are quite distinct as revealed by Figure 20. If t is the term $\lambda x.\lambda y.\lambda z.z$, in the case of the λ-term, the redex is at the bottom of the syntax tree but in the case of the $\overline{\lambda}\mu\tilde{\mu}$-term, it is at the top. This means that β-reduction would, in general, require a search at an arbitrary depth into the λ-term but not in the $\overline{\lambda}\mu\tilde{\mu}$-term This is shown in the following two reduction sequences, where the redex performed at each step is underlined:

$$\begin{array}{ll} \underline{(\lambda x.\lambda y.\lambda z.z)u_1}\ u_2\ u_3 & \rightarrow \\ \underline{(\lambda y.\lambda z.z)u_2}\ u_3 & \rightarrow \\ \underline{(\lambda z.z)u_3} & \rightarrow \\ u_3 & \end{array}$$

$$\begin{array}{ll} \mu\alpha.\underline{\langle \lambda x.\lambda y.\lambda z.z \mid u_1 \cdot u_2 \cdot u_3 \cdot \alpha\rangle} & \rightarrow \\ \mu\alpha.\underline{\langle u_1 \mid \tilde{\mu}x.\langle \lambda y.\lambda z.z \mid u_2 \cdot u_3 \cdot \alpha\rangle\rangle} & \rightarrow \\ \mu\alpha.\underline{\langle \lambda y.\lambda z.z \mid u_2 \cdot u_3 \cdot \alpha\rangle} & \twoheadrightarrow \\ \mu\alpha.\underline{\langle \lambda z.z \mid u_3 \cdot \alpha\rangle} & \twoheadrightarrow \\ u_3 & \end{array}$$

Notice how in the $\overline{\lambda}\mu\tilde{\mu}$-calculus there is always a redex within a bounded distance from the top of the syntax tree.

As described by Ager *et al.* [2003], one can obtain a tail recursive evaluation by translating λ-terms into continuation-passing style (CPS). Our approach instead is to translate λ-terms into the $\overline{\lambda}\mu\tilde{\mu}$-calculus which uses continuations but avoids the explicit conversion to CPS.

Consider the λ-term $t\ u_1\ u_2\ u_3$ which can be directly embedded in the $\overline{\lambda}\mu\tilde{\mu}$-calculus as $\langle \mu\alpha.\langle \mu\alpha.\langle t \mid u_1 \cdot \alpha\rangle \mid u_2 \cdot \alpha\rangle \mid u_3 \cdot \alpha\rangle$. The embedding actually represents a translation from a proof in natural deduction to one in the sequent calculus. The resulting $\overline{\lambda}\mu\tilde{\mu}$-term reduces as follows:

$$\begin{array}{ll} \langle \mu\alpha.\langle \mu\alpha.\langle t \mid u_1 \cdot \alpha\rangle \mid u_2 \cdot \alpha\rangle \mid u_3 \cdot \alpha\rangle & \rightarrow \\ \langle \mu\alpha.\langle t \mid u_1 \cdot \alpha\rangle \mid u_2 \cdot u_3 \cdot \alpha\rangle & \rightarrow \\ \langle t \mid u_1 \cdot u_2 \cdot u_3 \cdot \alpha\rangle & \end{array}$$

where each reduction occurs at the top of the term. The final term has the structure shown on the right of Figure 20.

Indeed, the connection between CPS and abstract machines is quite deep. In particular, Streicher and Reus [1998] present a *rational reconstruction* of the Krivine abstract machine from the "natural" denotational continuation semantics of the λ-calculus.

5.2 The problem with substitution

The version of the $\overline{\lambda}\mu\tilde{\mu}$-calculus presented above uses symbols to refer to variable names and includes substitution as a meta-operation. In typical abstract machines names are replaced by indices and substitution is typically expressed at a lower-level using calculi with explicit substitutions [Abadi et al. 1990]. Consider such an extension for $\overline{\lambda}\mu\tilde{\mu}$ with terms such as:

$$\underline{t[x \leftarrow u_1]}[y \leftarrow u_2][z \leftarrow u_3]$$

The problem with such an addition is that one loses the capability of always reducing at a bounded distance. In fact, the above potential redex (the one underlined)

Terms	v	$::=$	$r \mid \lambda r.v \mid \mu k.c \mid v\tau \mid v\ w$
Contexts	e	$::=$	$\mathtt{tp} \mid k \mid v \cdot e \mid \tilde{\mu} r.c \mid e\tau \mid e\ w$
Commands	c	$::=$	$\langle v \mid e \rangle \mid c\tau \mid c\ w$
Continuations Vars	k	$::=$	$\alpha \mid \gamma$
Substitutions	τ	$::=$	$[r \leftarrow v] \mid [k \leftarrow e]$
Weakenings	w	$::=$	$\uparrow^r \mid \uparrow^k$

Fig. 21. Syntax of the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-calculus

is now buried within the syntax tree instead of being at the top. Since there is no limit on the number of substitutions, we may have sequences such as:

$$\underline{t[x_1 \leftarrow u_1]}[x_2 \leftarrow u_2]\cdots[x_n \leftarrow u_n]$$

where the depth of the redex cannot be determined statically.

In other words, a direct extension of $\overline{\lambda}\mu\tilde{\mu}$ with explicit substitutions requires unbounded search for redexes. To avoid this unbounded search, the calculus should be extended with a notion of *simultaneous substitution*, which directly encodes the environment used by an abstract machine. This is for example the approach followed in the $\lambda\sigma_w$-calculus [Hardin et al. 1996], in which the above term becomes

$$t\ [u_1.u_2.u_3.\mathtt{id}]$$

where we have assumed the variables x, y and z correspond to the de Bruijn numbers 1, 2 and 3, respectively.

Thus, in order to get closer to the level of abstract machines, we modify the $\overline{\lambda}\mu\tilde{\mu}$-calculus by moving to de Bruijn indices and by including environments. However, unlike the various calculi for explicit simultaneous substitutions, environments need not be modeled as a new primitive notion in the $\overline{\lambda}\mu\tilde{\mu}$-calculus: just as lists of arguments are modeled using contexts, *environments too can be modeled using contexts.* In practice, this idea corresponds to the usual technique of maintaining both the arguments and the environment on the run-time stack [Douence and Fradet 1998].

6. THE $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-CALCULUS

We present the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-calculus: its syntax, types, and reductions. We then illustrate how both the call-by-name and call-by-value semantics can be defined using reduction sequences which always apply reductions within a bounded distance from the root.

6.1 Well-formed Terms and Types

The $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-calculus builds on the $\overline{\lambda}\mu\tilde{\mu}$-calculus with explicit substitution by adding explicit weakening. Its syntax is given in Figure 21. As in the $\overline{\lambda}\mu\tilde{\mu}$-calculus, the syntactic categories v, e and c correspond to terms, contexts and commands, respectively. There is however only one term variable r which corresponds to a fixed register (the accumulator). There are also only two continuation variables, α and γ, which allow one to simulate the environment and the working stack of arguments, respectively. We also admit a continuation constant called $\mathtt{tp}$ which denotes the top-level. One does not have an unbounded number of consecutive weakenings or

$$\frac{}{\mid k \;\vdash\; k}\;Axiom_l \qquad \frac{}{r \;\vdash\; r \mid}\;Axiom_r \qquad \frac{}{\mid \mathsf{tp} \;\vdash}\;Axiom_{\mathsf{tp}}$$

$$\frac{\Gamma \;\vdash\; v \mid \Delta \quad \Gamma \mid e \;\vdash\; \Delta}{\Gamma \mid v \cdot e \;\vdash\; \Delta}\;\rightarrow_l \qquad \frac{r \;\vdash\; v \mid \Delta}{\vdash\; \lambda r.v \mid \Delta}\;\rightarrow_r$$

$$\frac{c \;:\; (r \;\vdash\; \Delta)}{\mid \tilde{\mu} r.c \;\vdash\; \Delta}\;Activate_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; k, \Delta)}{\Gamma \;\vdash\; \mu k.c \mid \Delta}\;Activate_r$$

$$\frac{\Gamma \;\vdash\; v \mid \Delta \quad \Gamma \mid e \;\vdash\; \Delta}{\langle v \mid e\rangle \;:\; (\Gamma \;\vdash\; \Delta)}\;Cut$$

Substitution rules:

$$\frac{r \;\vdash\; v \mid \Delta \qquad \vdash\; v' \mid \Delta}{\vdash\; v[r \leftarrow v'] \mid \Delta}\;Sv_l \qquad \frac{\Gamma \;\vdash\; v \mid \Delta, k \quad \Gamma \mid e \;\vdash\; \Delta}{\Gamma \;\vdash\; v[k \leftarrow e] \mid \Delta}\;Sv_r$$

$$\frac{r \mid e \;\vdash\; \Delta \qquad \vdash\; v \mid \Delta}{\mid e[r \leftarrow v] \;\vdash\; \Delta}\;Se_l \qquad \frac{\Gamma \mid e \;\vdash\; \Delta, k \quad \Gamma \mid e' \;\vdash\; \Delta}{\Gamma \mid e[k \leftarrow e'] \;\vdash\; \Delta}\;Se_r$$

$$\frac{c \;:\; (r \;\vdash\; \Delta) \qquad \vdash\; v \mid \Delta}{c[r \leftarrow v] \;:\; (\;\vdash\; \Delta)}\;Sc_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; \Delta, k) \quad \Gamma \mid e \;\vdash\; \Delta}{c[k \leftarrow e] \;:\; (\Gamma \;\vdash\; \Delta)}\;Sc_r$$

Weakening rules:

$$\frac{\vdash\; v \mid \Delta}{r \;\vdash\; v \uparrow^r \mid \Delta}\;Wv_l \qquad \frac{\Gamma \;\vdash\; v \mid \Delta}{\Gamma \;\vdash\; v \uparrow^k \mid \Delta, k}\;Wv_r$$

$$\frac{\mid e \;\vdash\; \Delta}{r \mid e \uparrow^r \;\vdash\; \Delta}\;We_l \qquad \frac{\Gamma \mid e \;\vdash\; \Delta}{\Gamma \mid e \uparrow^k \;\vdash\; \Delta, k}\;We_r$$

$$\frac{c \;:\; (\Gamma \;\vdash\; \Delta)}{c \uparrow^r \;:\; (r \;\vdash\; \Delta)}\;Wc_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; \Delta)}{c \uparrow^k \;:\; (\Gamma \;\vdash\; \Delta, k)}\;Wc_r$$

Fig. 22. Well-formed $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ terms, contexts and commands

substitutions. There are at most three consecutive weakenings or substitutions. To express these syntactic restrictions we define in Figure 22 the notions of well-formed terms, contexts and commands using the following three distinct judgments:

$$\Gamma \;\vdash\; v \mid \Delta \qquad \Gamma \mid e \;\vdash\; \Delta \qquad c \;:\; (\Gamma \;\vdash\; \Delta)$$

Γ contains at most one term variable. The set Δ is restricted to at most two distinct continuation variables. The sets Γ and Δ correspond to the set of free term and continuation variables, respectively, occurring in either v, e or c. The function $\mathtt{fv}^r$

which determines if r occurs free in a term can be defined as following:

$$\begin{array}{lcl}
\texttt{fv}^r(r) & = & \{r\} \\
\texttt{fv}^r(\lambda r.v) & = & \{\} \\
\texttt{fv}^r(\mu k.c) & = & \texttt{fv}^r(c) \\
\texttt{fv}^r(v[r \leftarrow v']) & = & \{\} \\
\texttt{fv}^r(v[k \leftarrow e]) & = & \texttt{fv}^r(v) \\
\texttt{fv}^r(v \uparrow^r) & = & \{r\} \\
\texttt{fv}^r(v \uparrow^k) & = & \texttt{fv}^r(v)
\end{array}$$

The $\texttt{fv}^r$ on commands and contexts, and the function which determines the free continuation variables are defined in a similar way. For example, the following judgments are not derivable:

$$r \vdash \lambda r.r \mid \qquad \mid \texttt{tp} \vdash \alpha$$

since r does not occur free in $\lambda r.r$ and α does not occur free in $\texttt{tp}$. When we write $\Delta, k : A$ we assume k does not occur in Δ.

The reading of the axioms is as follows: term variable r (continuation variable k) is well-formed if it occurs in the set of free term (continuation) variables. Also, the constant $\texttt{tp}$ is well-formed. Notice that the axioms do not have redundant assumptions. For example, none of the following judgments is derivable:

$$\frac{}{r \vdash r \mid \alpha} \qquad \frac{}{r \mid \alpha \vdash \alpha} \qquad \frac{}{r \mid \alpha \vdash \alpha, \gamma}$$

They can however be obtained by applying weakening steps:

$$\dfrac{\dfrac{}{r \vdash r \mid}\,Axiom_r}{r \vdash r \uparrow^\alpha \mid \alpha}\,Wv_r \qquad \dfrac{\dfrac{\dfrac{}{\mid \alpha \vdash \alpha}\,Axiom_l}{r \mid \alpha \uparrow^r \vdash \alpha}\,We_l}{r \mid \alpha \uparrow^r \uparrow^\gamma \vdash \alpha, \gamma}\,We_r$$

As shown below, $(\lambda r.r)$ is a well-form term:

$$\dfrac{\dfrac{}{r \vdash r \mid}\,Axiom_r}{\vdash \lambda r.r \mid}\,\rightarrow_r$$

Notice that in the conclusion the set of assumptions is empty. To derive judgments with additional assumptions, explicit weakening steps must be used. For example, we can derive the following three judgments:

$$r \vdash (\lambda r.r) \uparrow^r \mid \qquad \vdash (\lambda r.r) \uparrow^\alpha \mid \alpha \qquad r \vdash (\lambda r.r) \uparrow^r \uparrow^\alpha \uparrow^\gamma \mid \alpha, \gamma$$

The term $\lambda r.\lambda r.r$ is not well-formed, as one cannot infer $r \vdash \lambda r.r \mid$. However, $\lambda r.((\lambda r.r) \uparrow^r)$ is well-formed:

$$\dfrac{\dfrac{\vdash \lambda r.r \mid}{r \vdash (\lambda r.r) \uparrow^r \mid}\,Wv_l}{\vdash \lambda r.(\lambda r.r) \uparrow^r \mid}\,\rightarrow_r$$

Notice that r occurs free in $(\lambda r.r) \uparrow^r$.

On the surface, the left implication rule does not seem to impose any restrictions. However, even a simple context such as $r \cdot \alpha$ is not well-formed. The reason is that both the argument and the rest of the context (α in this case) need to have the same set of free term and continuation variables. To that end, weakenings are added to the parameter and the rest of the list as shown below:

$$\dfrac{\dfrac{\dfrac{}{r \vdash r \mid}\,Axiom_r}{r \vdash r \uparrow^{\alpha} \mid \alpha}\,Wv_r \qquad \dfrac{\dfrac{}{\mid \alpha \vdash \alpha}\,Axiom_l}{r \mid \alpha \uparrow^{r} \vdash \alpha}\,We_l}{r \mid r \uparrow^{\alpha} \cdot \alpha \uparrow^{r} \vdash \alpha}\,\rightarrow_l$$

For the same reason, the context $(\lambda r.r) \cdot \alpha$ is not well-formed. One would need to write $(\lambda r.r) \uparrow^{\alpha} \cdot\alpha$. Like for the lambda rule, in the $\tilde{\mu}$ rule, variable r needs to occur free in the body. This explains the presence of the weakening in the term $\tilde{\mu}r.(\langle \lambda r.r \mid \mathsf{tp}\rangle \uparrow^{r})$. The same holds for the dual μ construct. Thus, one writes $\mu\alpha.(\langle \lambda r.r \mid \mathsf{tp}\rangle \uparrow^{\alpha})$. In the command rule, both the term and the context share the same set of assumptions. Thus, for example a command of the form $\langle r \mid \alpha\rangle$ is not well-formed. The command becomes well-formed after adding the appropriate weakenings: $\langle r \uparrow^{\alpha} \mid \alpha \uparrow^{r}\rangle$.

The substitution removes an assumption either on the left-hand side or the right-hand side of the sequent. Thus, $r[r \leftarrow v]$ has an empty Γ. According to the inference rule for $v[r \leftarrow v']$, v' is checked in an empty Γ. This means that v' cannot refer to r. For example, the term $r[r \leftarrow r]$ is not well-formed. The number of consecutive substitutions is at most three. If we were to allow circular substitutions, then this property would not hold since we could have an unbounded number of substitutions: $r[r \leftarrow r][r \leftarrow r]\cdots$. In applying a substitution to a term, the sets Γ and Δ have to be the same as the ones needed to check the term itself. The same holds for a context or a command. Thus, the term $r[r \leftarrow \mu\gamma.\langle(\lambda r.r) \uparrow^{\alpha} \mid \alpha\rangle]$ is not well-formed. One has to write $r \uparrow^{\alpha} [r \leftarrow \mu\gamma.\langle(\lambda r.r) \uparrow^{\alpha}\uparrow^{\gamma} \mid \alpha \uparrow^{\gamma}\rangle]$. Also, the term $(\lambda r.r)[r \leftarrow v]$ is not well-formed. One would need to write $(\lambda r.r) \uparrow^{r} [r \leftarrow v]$ as shown below (assume v is closed with respect to term and continuation variables):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{r \vdash r \mid}\,Axiom_r}{\vdash \lambda r.r \mid}\,\rightarrow_r}{r \vdash (\lambda r.r) \uparrow^{r} \mid}\,Wv_l \qquad \vdash v \mid}{\vdash (\lambda r.r) \uparrow^{r} [r \leftarrow v] \mid}\,Sv_l$$

A weakening step introduces one more assumption, either on the left-hand side or on the right-hand side of the sequent. Since Γ and Δ are restricted to at most one and two variables, respectively, one has at most three consecutive weakening steps. Moreover, the weakenings have to be distinct. For example, $(\lambda r.r) \uparrow^{r}\uparrow^{r}$ is not well-formed as $\vdash (\lambda r.r) \uparrow^{r}$ is not derivable. Analogously, the context $\gamma \uparrow^{\alpha}\uparrow^{\alpha}$ is not well-formed. In a term $v \uparrow^{\alpha}$, the weakening indicates that α does not occur free in v. The same holds for a weakening on a context or a command.

Proposition 1. *Given*

$$\Gamma \vdash v \mid \Delta \qquad \Gamma \mid e \vdash \Delta \qquad c : (\Gamma \vdash \Delta)$$

then

(1) Γ is non-empty iff r occurs free in v, e or c;

$$\frac{}{\mid k : A \;\vdash\; k : A}\,\mathit{Axiom}_l \qquad \frac{}{r : A \;\vdash\; r : A \mid}\,\mathit{Axiom}_r \qquad \frac{}{\mid \mathsf{tp} : A \;\vdash}\,\mathit{Axiom}_{\mathsf{tp}}$$

$$\frac{\Gamma \;\vdash\; v : A \mid \Delta \quad \Gamma \mid e : B \;\vdash\; \Delta}{\Gamma \mid v \cdot e : A \to B \;\vdash\; \Delta}\,\to_l \qquad \frac{r : A \;\vdash\; v : B \mid \Delta}{\vdash\; \lambda r.v : A \to B \mid \Delta}\,\to_r$$

$$\frac{c \;:\; (r : A \;\vdash\; \Delta)}{\mid \tilde{\mu} r.c : A \;\vdash\; \Delta}\,\mathit{Activate}_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; k : A, \Delta)}{\Gamma \;\vdash\; \mu k.c : A \mid \Delta}\,\mathit{Activate}_r$$

$$\frac{\Gamma \;\vdash\; v : A \mid \Delta \quad \Gamma \mid e : A \;\vdash\; \Delta}{\langle v \mid e \rangle \;:\; (\Gamma \;\vdash\; \Delta)}\,\mathit{Cut}$$

Substitution rule:

$$\frac{r : B \;\vdash\; v : A \mid \Delta \qquad \vdash\; v' : B \mid \Delta}{\vdash\; v[r \leftarrow v'] : A \mid \Delta}\,\mathit{Sv}_l \qquad \frac{\Gamma \;\vdash\; v : A \mid \Delta, k : B \quad \Gamma \mid e : B \;\vdash\; \Delta}{\Gamma \;\vdash\; v[k \leftarrow e] : A \mid \Delta}\,\mathit{Sv}_r$$

$$\frac{r : B \mid e : A \;\vdash\; \Delta \qquad \vdash\; v : B \mid \Delta}{\mid e[r \leftarrow v] : A \;\vdash\; \Delta}\,\mathit{Se}_l \qquad \frac{\Gamma \mid e : A \;\vdash\; \Delta, k : B \quad \Gamma \mid e' : B \;\vdash\; \Delta}{\Gamma \mid e[k \leftarrow e'] : A \;\vdash\; \Delta}\,\mathit{Se}_r$$

$$\frac{c \;:\; (r : B \;\vdash\; \Delta) \qquad \vdash\; v : B \mid \Delta}{c[r \leftarrow v] \;:\; (\;\vdash\; \Delta)}\,\mathit{Sc}_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; \Delta, k : B) \quad \Gamma \mid e : B \;\vdash\; \Delta}{c[k \leftarrow e] \;:\; (\Gamma \;\vdash\; \Delta)}\,\mathit{Sc}_r$$

Weakening rules:

$$\frac{\vdash\; v : A \mid \Delta}{r : B \;\vdash\; v \uparrow^r : A \mid \Delta}\,\mathit{Wv}_l \qquad \frac{\Gamma \;\vdash\; v : A \mid \Delta}{\Gamma \;\vdash\; v \uparrow^k : A \mid \Delta, k : B}\,\mathit{Wv}_r$$

$$\frac{\mid e : A \;\vdash\; \Delta}{r : B \mid e \uparrow^r : A \;\vdash\; \Delta}\,\mathit{We}_l \qquad \frac{\Gamma \mid e : A \;\vdash\; \Delta}{\Gamma \mid e \uparrow^k : A \;\vdash\; \Delta, k : B}\,\mathit{We}_r$$

$$\frac{c \;:\; (\Gamma \;\vdash\; \Delta)}{c \uparrow^r \;:\; (r : B \;\vdash\; \Delta)}\,\mathit{Wc}_l \qquad \frac{c \;:\; (\Gamma \;\vdash\; \Delta)}{c \uparrow^k \;:\; (\Gamma \;\vdash\; \Delta, k : B)}\,\mathit{Wc}_r$$

Fig. 23. Type system for the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ calculus

(2) Δ corresponds to the set of free continuation variables occurring in v, e or c.

The type system is in Figure 23. It reflects the well-formedness conditions explained in the previous section (well-typed terms are well-formed), but is otherwise straightforward.

6.2 Reduction Semantics

The reduction semantics is given in Figures 24, 25 and 26. We first explain the computational rules. The β-rule needs a weakening of the context. Otherwise, the rule would produce a term which is not well-formed since r does not occur free in e. The rules μ and $\tilde{\mu}$ do not need any additional weakenings. Take the μ rule for example, for the left-hand side to be well-formed continuation variable k

$$
\begin{array}{lrcl}
(\beta) & \langle \lambda r.v \mid v' \cdot e \rangle & \to & \langle v' \mid \tilde{\mu} r.\langle v \mid e \uparrow^r \rangle \rangle \\
(\mu) & \langle \mu k.c \mid e \rangle & \to & c[k \leftarrow e] \\
(\tilde{\mu}) & \langle v \mid \tilde{\mu} r.c \rangle & \to & c[r \leftarrow v]
\end{array}
$$

Fig. 24. Computational rules for $\lambda\mu\tilde{\mu}\mathtt{r}{\uparrow}$

$$
\begin{array}{lrcl}
(sv) & \mu k.\langle v \uparrow^k \mid k\ w^v \rangle & \to & v \\
(se) & \tilde{\mu} r.\langle r\ w^e \mid e \uparrow^r \rangle & \to & e
\end{array}
$$

Fig. 25. Simplification rules for $\lambda\mu\tilde{\mu}\mathtt{r}{\uparrow}$

Rules on terms

$$
\begin{array}{lrcl}
(r\tau) & r[r \leftarrow v] & \to & v \\
(\lambda\tau) & (\lambda r.v)[k \leftarrow e] & \to & \lambda r.(v[k \leftarrow e \uparrow^r]) \\
(\mu\tau 1) & (\mu k.c)[r \leftarrow v] & \to & \mu k.(c[r \leftarrow v \uparrow^k]) \\
(\mu\tau 2) & (\mu k.c)[k' \leftarrow e] & \to & \mu k.(c[k' \leftarrow e \uparrow^k]) \\
(v \uparrow^r 1) & (v \uparrow^r)[r \leftarrow v'] & \to & v \\
(v \uparrow^k 1) & (v \uparrow^k)[k \leftarrow e] & \to & v \\
(v \uparrow^r 2) & (v \uparrow^r)[k \leftarrow e \Uparrow^r] & \to & (v[k \leftarrow e \Uparrow]) \uparrow^r \\
(v \uparrow^k 2) & (v \uparrow^k)[k' \leftarrow e \Uparrow^k] & \to & (v[k' \leftarrow e \Uparrow]) \uparrow^k \\
(v \uparrow^k 3) & (v \uparrow^k)[r \leftarrow v' \Uparrow^k] & \to & (v[r \leftarrow v' \Uparrow]) \uparrow^k
\end{array}
$$

Rules on contexts

$$
\begin{array}{lrcl}
(k\tau) & k[k \leftarrow e] & \to & e \\
(\cdot\tau) & (v \cdot e)\tau & \to & (v\tau) \cdot (e\tau) \\
(\tilde{\mu}\tau) & (\tilde{\mu} r.c)[k \leftarrow e] & \to & \tilde{\mu} r.(c[k \leftarrow e \uparrow^r]) \\
(e \uparrow^r 1) & (e \uparrow^r)[r \leftarrow v] & \to & e \\
(e \uparrow^k 1) & (e \uparrow^k)[k \leftarrow e'] & \to & e \\
(e \uparrow^r 2) & (e \uparrow^r)[k \leftarrow e' \Uparrow^r] & \to & (e[k \leftarrow e' \Uparrow]) \uparrow^r \\
(e \uparrow^k 2) & (e \uparrow^k)[k' \leftarrow e' \Uparrow^k] & \to & (e[k' \leftarrow e' \Uparrow]) \uparrow^k \\
(e \uparrow^k 3) & (e \uparrow^k)[r \leftarrow v \Uparrow^k] & \to & (e[r \leftarrow v \Uparrow]) \uparrow^k
\end{array}
$$

Rules on commands

$$
\begin{array}{lrcl}
(c\tau) & \langle v \mid e \rangle\tau & \to & \langle v\tau \mid e\tau \rangle \\
(c \uparrow^r 1) & (c \uparrow^r)[r \leftarrow v] & \to & c \\
(c \uparrow^k 1) & (c \uparrow^k)[k \leftarrow e] & \to & c \\
(c \uparrow^r 2) & (c \uparrow^r)[k \leftarrow e \Uparrow^r] & \to & (c[k \leftarrow e \Uparrow]) \uparrow^r \\
(c \uparrow^k 2) & (c \uparrow^k)[k' \leftarrow e \Uparrow^k] & \to & (c[k' \leftarrow e \Uparrow]) \uparrow^k \\
(c \uparrow^k 3) & (c \uparrow^k)[r \leftarrow v \Uparrow^k] & \to & (c[r \leftarrow v \Uparrow]) \uparrow^k
\end{array}
$$

Fig. 26. Substitution rules for $\lambda\mu\tilde{\mu}\mathtt{r}{\uparrow}$

has to occur free in c and cannot occur free in e. This makes the right-hand side well-formed.

The simplification rules of Figure 25 make use of the notation w^v and w^e which denotes a sequence of weakenings corresponding to the free variables in v and e, respectively. For example, if e is α then w^e would be $\uparrow^\alpha$. The weakening applied to term v occurring in the left-hand side of the sv-rule captures the fact that k does not occur free in v. Usually this is expressed using a proviso. For example, in $\lambda\mathcal{C}$ the rule becomes:

$$\mathcal{C}(\lambda k.k\ M) \to M \qquad k \text{ not free in } M$$

where $k\ M$ corresponds to $\langle M \mid k\rangle$. The weakenings on k are necessary to deal with the case that v has free variables. Otherwise, the following reduction would not be possible:

$$\mu k.\langle r \uparrow^k \ \mid k \uparrow^r\rangle \to r$$

Analogously, rule se contains the weakening on r to indicate that r cannot occur free in e. It needs the weakenings on r to cover the case that e has free continuation variables. For example, one has:

$$\langle v \mid \tilde{\mu} r.\langle r \uparrow^\alpha \ \mid \alpha \uparrow^r\rangle\rangle \to \langle v \mid \alpha\rangle$$

Instead of binding r to v and then invoking α with r, one invokes α directly with v.

The rules of Figure 26 move substitutions to the leaves and then use them to look up the values of variables. We explain the rules on terms. The $r\tau$-rule applies the substitution $[r \leftarrow v]$ to variable r. One might also be inclined to introduce the rule $r[k \leftarrow e] \to r$. This however is not correct since $r[k \leftarrow e]$ is not well-formed. To make it well-formed one should write $(r \uparrow^k)[k \leftarrow e \uparrow^r]$ which by application of $(v \uparrow^k 1)$ reduces to r. The following three rules move a substitution across a λ and a μ. Since we do not have α-renaming in our calculus, these rules might seem suspicious. For example, moving a substitution inside a λ might cause a free occurrence of the accumulator to get bound. This however does not occur in our setting. For the term $(\lambda r.v)[k \leftarrow e]$ to be well-formed variable r cannot occur free in e. Analogously, in the $\mu\tau 1$ and $\mu\tau 2$ rules, k cannot occur free in v and e, respectively. Therefore, no variable capture can occur. However, to move a substitution inside a binder one still needs to apply some weakening to the term or context to be substituted. Otherwise, a reduction might produce a term that is not well-formed:

$$(\lambda r.r \uparrow^\alpha)[\alpha \leftarrow \mathsf{tp}] \to \lambda r.(r \uparrow^\alpha [\alpha \leftarrow \mathsf{tp}])$$

With the appropriate weakening, the reduction produces a well-formed term:

$$\lambda r.(r \uparrow^\alpha [\alpha \leftarrow \mathsf{tp} \uparrow^r])$$

This explains the weakenings applied to the right-hand sides of the $\lambda\tau$, $\mu\tau 1$ and $\mu\tau 2$ rules. There is no reduction for a term of the form $(\mu k.c)[k \leftarrow e]$ or $(\lambda r.v)[r \leftarrow v]$ as they are not well-formed. Rules $(v \uparrow^r 1)$ and $(v \uparrow^k 1)$ illustrate how weakening corresponds to explicit memory deallocation. Since r and k do not occur free in v one can get rid of the substitution. The remaining three rules move a substitution across a weakening. This requires some care. For example, one might consider the following rule:

$$v \uparrow^r [k \leftarrow e] \rightarrow v[k \leftarrow e] \uparrow^r$$

For the left-hand side to be well-formed, r has to occur free in e, but this produces a term on the right-hand side which is not well-formed. The solution is to express the above reduction as follows:

$$v \uparrow^r [k \leftarrow e \Uparrow^r] \rightarrow v[k \leftarrow e \Uparrow] \uparrow^r$$

The notation $\Uparrow^r$ ($\Uparrow^k$) stands for a sequence of weakenings containing a weakening for r (k). If $\Uparrow^r$ ($\Uparrow^k$) occurs in the left-hand side of a rule, then the occurrence of $\Uparrow$ in the right-hand side stands for the same sequence of weakening minus the weakening on r (k). For example,

$$v \uparrow^r [\gamma \leftarrow e \uparrow^\alpha \uparrow^r] \rightarrow v[\gamma \leftarrow e \uparrow^\alpha] \uparrow^r$$

The rules on contexts and commands follow the same pattern. For example, as discussed above it is possible to move a substitution inside a $\tilde{\mu}$-construct, by applying the appropriate weakening on r. There is no rule for a term of the form $(\tilde{\mu}r.c)[r \leftarrow v]$ since is not well-formed.

Example 6. We show the reduction of

$$\mu\alpha.\langle \lambda r.(r \uparrow^\alpha) \mid v \uparrow^\alpha \cdot\alpha \rangle$$

where v is a closed term.

$$\begin{array}{ll}
\mu\alpha.\langle \lambda r.(r \uparrow^\alpha) \mid v \uparrow^\alpha \cdot\alpha \rangle & \rightarrow_\beta \\
\mu\alpha.\langle v \uparrow^\alpha \mid \tilde{\mu}r.\langle r \uparrow^\alpha \mid \alpha \uparrow^r \rangle \rangle & \rightarrow_{\tilde{\mu}} \\
\mu\alpha.(\langle r \uparrow^\alpha \mid \alpha \uparrow^r \rangle [r \leftarrow v \uparrow^\alpha]) & \rightarrow_{c\tau} \\
\mu\alpha.\langle r \uparrow^\alpha [r \leftarrow v \uparrow^\alpha] \mid \alpha \uparrow^r [r \leftarrow v \uparrow^\alpha] \rangle & \rightarrow_{v\uparrow^k 3} \\
\mu\alpha.\langle r[r \leftarrow v] \uparrow^\alpha \mid \alpha \uparrow^r [r \leftarrow v \uparrow^\alpha] \rangle & \rightarrow_{r\tau} \\
\mu\alpha.\langle v \uparrow^\alpha \mid \alpha \uparrow^r [r \leftarrow v \uparrow^\alpha] \rangle & \rightarrow_{e\uparrow^k 1} \\
\mu\alpha.\langle v \uparrow^\alpha \mid \alpha \rangle & \rightarrow_{sv} \\
v &
\end{array}$$

Proposition 2.

(i) The $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ satisfies subject reduction. Let l be a command, term or context:
—if l is well-formed and $l \rightarrow l'$ then l' is well-formed;
—if l is well-typed and $l \rightarrow l'$ then l' is well-typed;

(ii) The $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ satisfies strong normalization.

Proof. The proof strategy for strong normalization has been suggested to us by Emmanuel Polonovski, who developed a general framework for proving strong normalization for calculi with explicit substitutions. It consists of first showing strong normalization for the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ without explicit substitution (but with weakening) by applying a variant of the reducibility technique. Next, the PSN (preservation of strong normalization) property is shown: if a term strongly normalizes in the calculus without explicit substitution, it also does in the one with explicit substitution. Finally, an embedding of a $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ term into a term without explicit substitutions is shown. The explicit substitutions are turned into redexes. The strong normalization of the whole calculus follows then from the PSN property. The proofs mirror the ones given by Polonovski [2004]. □

6.3 Call-by-name and call-by-value evaluation

Before formalizing the call-by-name and call-by-value evaluation of the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$, we present in Figures 27, 28 and 29 three reduction strategies: $\mapsto_v$, $\mapsto_e$ and $\mapsto_c$. They make use of the following notation: $\rightarrow_v$ denotes the substitution rules on terms together with the sv simplification rule; $\rightarrow_e$ denotes the substitution rules on contexts together with the se simplification rule and finally $\rightarrow_c$ denotes the substitution rules on commands.

The three strategies make use of the auxiliary reductions $\mapsto_{v\leq 2\tau}$, $\mapsto_{e\leq 2\tau}$ and $\mapsto_{c\leq 2\tau}$ whose goal is to apply $\rightarrow_v$, $\rightarrow_e$ or $\rightarrow_c$ under at most two substitutions. Moreover, they go under a substitution only if the outermost term, context or command is not a redex. For example, a term of the form

$$\underline{\underline{r[r \leftarrow v]} \uparrow^{\alpha} [\alpha \leftarrow e]}$$

contains two $\rightarrow_v$-redexes: the outer redex is a $v \uparrow^k 1$-redex and the inner redex is a $r\tau$-redex. The strategy $\mapsto_{v\leq 2\tau}$ will reduce the outermost redex. The same happens in the following context and command:

$$\underline{\underline{\alpha[\alpha \leftarrow s]} \uparrow^{r} [r \leftarrow v]} \qquad \underline{\underline{\langle v \mid e\rangle\tau} \uparrow^{\alpha} [\alpha \leftarrow s]}$$

The strategies $\mapsto_{e\leq 2\tau}$ and $\mapsto_{c\leq 2\tau}$ will reduce the $(e \uparrow^r 1)$-redex and the $(c \uparrow^k 1)$-redex, respectively.

The strategy $\mapsto_v$ simply invokes $\mapsto_{v\leq 2\tau}$. In addition to invoking $\mapsto_{e\leq 2\tau}$, the $\mapsto_e$ strategy reduces a context of the form $v \cdot e$ in a left-to-right fashion. First, the strategy $\mapsto_v$ is applied to v. Next, the strategy $\mapsto_{e\leq 2\tau}$ is applied to e. This means that a deallocation step is carried out before any other reduction. For example,

$$r[r \leftarrow v] \cdot e \mapsto_e v \cdot e \qquad \text{and} \qquad (\lambda r.r) \cdot \alpha[\alpha \leftarrow s] \mapsto_e (\lambda r.r) \cdot s$$

However,

$$v \cdot r[r \leftarrow v'] \cdot e \not\mapsto_e v \cdot v' \cdot e$$

since the strategy $\mapsto_{e\leq 2\tau}$ only goes under substitutions and not a context $v \cdot e$.

The strategy $\mapsto_c$ in addition to the invocation of $\mapsto_{c\leq 2\tau}$, it first applies the strategy $\mapsto_e$ to the consumer, next the strategy $\mapsto_v$ is applied to the producer.

Finally, the call-by-name and call-by-value strategies are given in Figures 30 and 31, respectively.[1] As described by Curien and Herbelin [2000], they resolve the critical pair in a command of the form $\langle \mu k.c \mid \tilde{\mu} r.c' \rangle$ in favor of the producer, for call-by-value, and in favor of the consumer for call-by-name.

PROPOSITION 3. *The call-by-name and call-by-value reduction strategies for the well-formed $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus are such that the reduction steps are confined to a bounded distance from the top of the syntax tree.*

[1]The description follows closely the implementation that can be found at `http://www.cs.indiana.edu/~sabry/papers/sequent_code.tar`.

$$\frac{v \rightarrow_v v'}{v \mapsto_{v \leq 2\tau} v'}$$

$$\frac{v \rightarrow_v v' \quad v\tau_1 \text{ not a } \rightarrow_v\text{-redex}}{v\tau_1 \mapsto_{v \leq 2\tau} v'\tau_1} \qquad \frac{v \rightarrow_v v' \quad v\tau_1\tau_2 \text{ and } v\tau_1 \text{ not } \rightarrow_v\text{-redexes}}{v\tau_1\tau_2 \mapsto_{v \leq 2\tau} v'\tau_1\tau_2}$$

$$\frac{v \mapsto_{v \leq 2\tau} v'}{v \mapsto_v v'}$$

Fig. 27. The strategy $\mapsto_v$

$$\frac{e \rightarrow_e e'}{e \mapsto_{e \leq 2\tau} e'}$$

$$\frac{e \rightarrow_e e' \quad e\tau_1 \text{ not a } \rightarrow_e\text{-redex}}{e\tau_1 \mapsto_{e \leq 2\tau} e'\tau_1} \qquad \frac{e \rightarrow_e e' \quad e\tau_1\tau_2 \text{ and } e\tau_1 \text{ not } \rightarrow_e\text{-redexes}}{e\tau_1\tau_2 \mapsto_{e \leq 2\tau} e'\tau_1\tau_2}$$

$$\frac{e \mapsto_{e \leq 2\tau} e'}{e \mapsto_e e'}$$

$$\frac{v \mapsto_v v'}{v \cdot e \mapsto_e v' \cdot e} \qquad \frac{e \mapsto_{e \leq 2\tau} e' \text{ and } v_{gc} \text{ not a } \mapsto_v\text{-redex}}{v_{gc} \cdot e \mapsto_e v_{gc} \cdot e'}$$

Fig. 28. The strategy $\mapsto_e$

$$\frac{c \rightarrow_c c'}{c \mapsto_{c \leq 2\tau} c'}$$

$$\frac{c \rightarrow_c c' \quad c\tau_1 \text{ not a } \rightarrow_c\text{-redex}}{c\tau_1 \mapsto_{c \leq 2\tau} c'\tau_1} \qquad \frac{c \rightarrow_c c' \quad c\tau_1\tau_2 \text{ and } c\tau_1 \text{ not } \rightarrow_c\text{-redexes}}{c\tau_1\tau_2 \mapsto_{c \leq 2\tau} c'\tau_1\tau_2}$$

$$\frac{c \mapsto_{c \leq 2\tau} c'}{c \mapsto_c c'}$$

$$\frac{e \mapsto_e e'}{\langle v \mid e \rangle \mapsto_c \langle v \mid e' \rangle} \qquad \frac{v \mapsto_v v' \quad \text{ and } e_{gc} \text{ not a } \mapsto_e \text{-redex}}{\langle v \mid e_{gc} \rangle \mapsto_c \langle v' \mid e_{gc} \rangle}$$

Fig. 29. The strategy $\mapsto_c$

$$\frac{c \mapsto_c c'}{c \overset{cbn}{\longmapsto} c'}$$

Reduce by focusing on the consumer:

$\langle v_{gc} \mid \tilde{\mu}r.c\rangle$	$\overset{cbn}{\longmapsto}$	$c[r \leftarrow v_{gc}]$	v_{gc} not a $\mapsto_v$-redex
$\langle \mu k.c \mid e_{gc}\rangle$	$\overset{cbn}{\longmapsto}$	$c[k \leftarrow e_{gc}]$	$e_{gc} \neq \tilde{\mu}r.c'$ and not a $\mapsto_e$-redex
$\langle \lambda r.v \mid v_{gc} \cdot e_{gc}\rangle$	$\overset{cbn}{\longmapsto}$	$\langle v_{gc} \mid \tilde{\mu}r.\langle v \mid e_{gc} \uparrow^r\rangle\rangle$	$v_{gc} \cdot e_{gc}$ not a $\mapsto_e$-redex

Fig. 30. The call-by-name strategy : $\overset{cbn}{\longmapsto}$

$$\frac{c \mapsto_c c'}{c \overset{cbv}{\longmapsto} c'}$$

Reduce by focusing on the producer:

$\langle \mu k.c \mid e_{gc}\rangle$	$\overset{cbv}{\longmapsto}$	$c[k \leftarrow e_{gc}]$	e_{gc} not a $\mapsto_e$-redex
$\langle v_{gc} \mid \tilde{\mu}r.c\rangle$	$\overset{cbv}{\longmapsto}$	$c[r \leftarrow v_{gc}]$	$v_{gc} \neq \mu k.c'$ and not a $\mapsto_v$-redex
$\langle \lambda r.v \mid v_{gc} \cdot e_{gc}\rangle$	$\overset{cbv}{\longmapsto}$	$\langle v_{gc} \mid \tilde{\mu}r.\langle v \mid e_{gc} \uparrow^r\rangle\rangle$	$v_{gc} \cdot e_{gc}$ not a $\mapsto_e$-redex

Fig. 31. The call-by-value strategy : $\overset{cbv}{\longmapsto}$

PROOF. There are no inference rules that create a recursive definition of a relation. For both strategies, all the paths of the dependency graph are as follows:

$$\begin{array}{l}
\mapsto_c, \mapsto_e, \mapsto_v, \mapsto_{v\leq 2\tau}, \rightarrow_v \\
\mapsto_c, \mapsto_e, \mapsto_{e\leq 2\tau}, \rightarrow_e \\
\mapsto_c, \mapsto_v, \mapsto_{v\leq 2\tau}, \rightarrow_v \\
\mapsto_c, \mapsto_{c\leq 2\tau}, \rightarrow_c
\end{array}$$

Hence, all reductions specified by the reduction sequences are confined to a bounded distance from the top of the syntax tree.

□

A program and an answer are well-formed closed commands. Moreover, an answer is of the form $\langle \lambda r.v \mid \mathtt{tp}\rangle$.

LEMMA 1. *Let p be a program, and a, a' be answers. We have:*

(i) p reduces to an answer a in the call-by-name $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ if and only if there exists an answer a' such that $p \overset{cbn}{\longmapsto\!\!\!\!\!\rightarrow} a' \twoheadrightarrow a$;

(ii) p reduces to an answer a in the call-by-value $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ if and only if there exists an answer a' such that $p \overset{cbv}{\longmapsto\!\!\!\!\!\rightarrow} a' \twoheadrightarrow a$.

Proof. We follow the proof technique of Huet and Lévy [1991]. Let *cbn* be the call-by-name reduction $p \twoheadrightarrow a$. This reduction needs to contract the descendant of the standard redex, say u_1, occurring in p. One then constructs the projection of the *cbn*-reduction with respect to the u_1-reduction. We denote this reduction as cbn/u_1. Since the reduction cbn/u_1 also leads to an answer, one can proceed by performing the projection $(cbn/u_1)/u_2$, where u_2 is the standard redex contracted by the reduction cbn/u_1. As before, also $(cbn/u_1)/u_2$ leads to an answer. The termination of such a process is guaranteed by showing that at each step the weight associated to each reduction decreases. Pictorially:

$$\begin{tikzcd}
p \arrow[r, two heads, "cbn"] \arrow[d, "u_1"'] & a \\
p_1 \arrow[r, two heads, "cbn/u_1"] \arrow[d, "u_2"'] & a \\
p_2 \arrow[r, two heads, "(cbn/u_1)/u_2"] & a \\
\vdots \arrow[d, "u_n"'] & \\
a' \arrow[r, two heads, "(\cdots((cbn/u_1)/u_2)\cdots/u_n)"] & a
\end{tikzcd}$$

The reduction $p \rightarrow^{u_1} p_1 \rightarrow^{u_2} p_2 \cdots \rightarrow^{u_n} a'$ is the desired standard reduction. □

7. AN ABSTRACT INSTRUCTION SET FOR THE $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-CALCULUS

The $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus imposes a useful structure on terms that is closer to the level of an abstract machine. In this section we introduce a few "macros" over the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus which define an "abstract instruction set" similar to that of typical abstract machines. We illustrate how the λ_{DB} terms can be translated to this abstract instruction set. In the next section we show that the call-by-name evaluation of the abstract instruction set corresponds to the Krivine machine, and that its call-by-value evaluation corresponds to the CEK machine.

7.1 Instruction Set

The instructions and their definitions in terms of $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ terms are given in Figure 32. The definition of the instructions is complicated by the weakening steps which although essential for capturing the proper semantics obscure some of the basic ideas. The instructions correspond to commands containing context variables α and γ free. Moreover, the first four instructions have also r free. To understand the instructions, it is useful to think of the three special variables r, γ, and α

$$\begin{array}{llll}
\textit{Instructions} & \textit{ins} & ::= & \texttt{Exec} \mid \texttt{Clear} \mid \texttt{PushArg} \mid \texttt{Extend-env} \mid \\
 & & & \textsf{bind } \texttt{Closure}\ c \textsf{ in } c' \mid \texttt{PopArg} \mid \texttt{Lookup-env} \\
\textit{Code} & c & ::= & \textit{ins}; c \mid \textit{ins}
\end{array}$$

$$\begin{array}{ll}
\texttt{Exec} & = \langle r \uparrow^{\alpha}\uparrow^{\gamma} \mid \alpha \uparrow^{r}\uparrow^{\gamma}\rangle \\
(\texttt{Clear};\ c) & = c \uparrow^{r} \\
(\texttt{PushArg};\ c) & = \langle (\mu\alpha.c) \uparrow^{\alpha}\uparrow^{r} \mid (r \uparrow^{\alpha}\uparrow^{\gamma}) \cdot (\alpha \uparrow^{r}\uparrow^{\gamma})\rangle \\
(\texttt{Extend-env};\ c) & = \langle (\mu\gamma.c) \uparrow^{r}\uparrow^{\gamma} \mid (r \uparrow^{\alpha}\uparrow^{\gamma}) \cdot (\gamma \uparrow^{\alpha}\uparrow^{r})\rangle \\
(\textsf{bind } \texttt{Closure}\ c \textsf{ in } c') & = \langle (\mu\alpha.c) \uparrow^{\alpha} \mid \tilde{\mu} r.c'\rangle \\
(\texttt{PopArg};\ c) & = \langle (\lambda r.(\mu\alpha.c)) \uparrow^{\alpha} \mid \alpha \uparrow^{\gamma}\rangle \\
(\texttt{Lookup-env};\ c) & = \langle (\lambda r.(\mu\gamma.c)) \uparrow^{\gamma} \mid \gamma \uparrow^{\alpha}\rangle
\end{array}$$

Fig. 32. Instructions and their definitions

as three registers of an abstract machine corresponding to the accumulator, the current environment (frame pointer), and the current stack pointer.

—The `Exec` instruction executes the code in the accumulator r using the current stack α. This could be expressed with the command $\langle r \mid \alpha\rangle$, which however is not well-formed. To make it well-formed one has to add the appropriate weakenings, obtaining $\langle r \uparrow^{\alpha} \mid \alpha \uparrow^{r}\rangle$. The weakening on γ is added to maintain the invariant of having α and γ free.

—The `Clear` instruction corresponds to clearing the accumulator.

—The `PushArg` and `Extend-env` instructions correspond to re-binding either α to a new stack with r on top of it (*i.e.* the stack $r \cdot \alpha$) or re-binding γ to a new environment with r on top of it (*i.e.* the environment $r \cdot \gamma$).

—The bind `Closure` c in c' instruction sequence corresponds to $\langle \mu\alpha.c \mid \tilde{\mu} r.c'\rangle$. The static nature of the environment is captured by the static scope of variable γ. The dynamic nature of the working stack instead is captured by the fact that variable α is redefined in c, and is thus bounded at activation time.

—The `PopArg` and `Lookup-env` correspond to rebinding r and α or rebinding r and γ.

7.2 Translating λ_{DB}-terms

The compilation in Figure 33 maps λ_{DB} terms to the abstract instruction set defined in the previous section. Variable lookups are compiled to a sequence of instructions that traverse the environment the specified number of times in the de Bruijn index. For example, the compilation of $(\bullet\uparrow)\uparrow$ is

`Lookup-env; Clear; Lookup-env; Clear; Lookup-env; Exec`

The first two elements in the environment get thrown away. The third element gets executed.

A λ-abstraction is compiled to a sequence of instructions which once executed grabs the argument from the stack, pushes it on the environment, and then executes the body. For example, the compilation of $\lambda\lambda\bullet$ is:

`PopArg; Extend-env; PopArg; Extend-env; Lookup-env; Exec`

$$
\begin{array}{|lcl|}
\hline
[\![\bullet]\!] & = & \texttt{Lookup-env; Exec} \\
[\![t\uparrow]\!] & = & \texttt{Lookup-env; Clear; } [\![t]\!] \\
[\![\lambda t]\!] & = & \texttt{PopArg; Extend-env; } [\![t]\!] \\
[\![t_1\ t_2]\!] & = & \textsf{bind}\ \texttt{Closure}\ [\![t_2]\!]\ \textsf{in}\ \texttt{PushArg; } [\![t_1]\!] \\
\hline
\end{array}
$$

Fig. 33. Compilation of λ_{DB} into $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$

An application is compiled to instructions for building a closure of the argument, saving it on the stack, and then evaluating the term in the function position. At this point, it is unspecified if the closure corresponding to the argument is evaluated before the call or not. This depends on whether we choose the call-by-name or the call-by-value semantics.

PROPOSITION 4. *Given a λ_{DB} term t, $[\![t]\!]$ is a well-formed $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ command closed with respect to r:*

$$[\![t]\!]\ :\ (\ \vdash\ \alpha, \gamma)$$

7.3 Translating λ-terms

Had we started with the regular λ-calculus instead of the λ_{DB}-terms, the compilation of terms would have been slightly more complicated. In particular, the translation would have needed to be parameterized with respect to the sequence Γ. This is necessary since the compilation of $y : B, x : A\ \vdash\ x : A$ should be different from the compilation of $x : A, y : B\ \vdash\ x : A$. This is already taken care when using de Bruijn indices, since the two distinct judgments above correspond to $B, A\ \vdash\ \underline{1} : A$ and $A, B\ \vdash\ \underline{2} : A$, respectively.

8. THE $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-CALCULUS AND ABSTRACT MACHINES

As stated in Proposition 4, the compilation of a λ_{DB} term is not a program since it contains α and γ free. To make it a program one will have to initialize those variables. That is indeed what happens in the translation of an abstract machine. In this section we consider the (call-by-name) Krivine machine [Krivine 2007] and the (call-by-value) CEK machine [Felleisen and Friedman 1986]. Both machines are translated to well-formed $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ programs. The call-by-name evaluation of these programs corresponds to the execution of the Krivine machine (see Section 8.1). Whereas, the call-by-value evaluation corresponds to the right-to-left CEK machine (see Section 8.2). Both results are shown by making use of an intermediate machine.

8.1 Correspondence with the Krivine Machine

We establish the correspondence between the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ and the Krivine machine in two steps: we first give in Section 8.1.1 a call-by-name interpretation to the instruction set given above, this results in a call-by-name intermediate abstract machine. We relate this intermediate machine to the call-by-name strategy of the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ calculus. Next, in Section 8.1.2 we relate the Krivine machine to the call-by-name intermediate abstract machine. We thus obtain that the Krivine machine corresponds to the call-by-name reduction strategy of the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-calculus.

Syntax:

state	st	$::=$	$(c, e, s) \mid (c, e, s, v)$
environment	e	$::=$	$v \cdot e \mid \mathsf{tp}$
stack	s	$::=$	$v \cdot s \mid \mathsf{tp}$
closure	v	$::=$	$c[e]$

Well-formed states:

$$\frac{c \ : \ (r \ \vdash \ \alpha, \gamma) \quad \mid e \ \vdash \qquad \mid s \ \vdash \qquad \vdash \ v \mid}{(c, e, s, v) : \ \vdash}$$

$$\frac{c \ : \ (\ \vdash \ \alpha, \gamma) \quad \mid e \ \vdash \qquad \mid s \ \vdash}{(c, e, s) : \ \vdash}$$

$$\frac{}{\mid \mathsf{tp} \ \vdash}$$

$$\frac{\vdash \ v \mid \qquad \mid e \ \vdash}{\mid v \cdot e \ \vdash} \qquad \frac{\vdash \ v \mid \qquad \mid s \ \vdash}{\mid v \cdot s \ \vdash}$$

$$\frac{c \ : \ (\ \vdash \ \alpha, \gamma) \quad \mid e \ \vdash}{\vdash \ c[e] \mid}$$

Transition rules:

$(\texttt{Exec}, e, s, c[e'])$	$\rightarrow$	(c, e', s)
$(\texttt{Clear}; c, e, s, v)$	$\rightarrow$	(c, e, s)
$(\textsf{bind } \texttt{Closure } c \textsf{ in } c', e, s)$	$\rightarrow$	$(c', e, s, c[e])$
$(\texttt{PushArg}; c, e, s, v)$	$\rightarrow$	$(c, e, v \cdot s)$
$(\texttt{PopArg}; c, e, v \cdot s)$	$\rightarrow$	(c, e, s, v)
$(\texttt{Extend-env}; c, e, s, v)$	$\rightarrow$	$(c, v \cdot e, s)$
$(\texttt{Lookup-env}; c, v \cdot e, s)$	$\rightarrow$	(c, e, s, v)

Fig. 34. The call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine

8.1.1 *Call-by-Name Abstract Machine.* As shown in Figure 34, the call-by-name intermediate machine has two kinds of states:

$$(c, e, s) \qquad \text{and} \qquad (c, e, s, v)$$

where c stands for the code or sequence of instructions as introduced in Figure 32; e and s stand for the environment and stack, respectively, and v is the accumulator. The basic value manipulated by the machine is a *closure* which as usual consists of code and an environment. The accumulator holds a closure. Both the environment and the stack are sequences of closures ending with the top-level continuation.

(c, e, s, v) is a well-formed configuration (written $(c, e, s, v) : \ \vdash \ $) if c is expecting something in the accumulator, in other words, r occurs free in c. Likewise, (c, e, s)

$$\boxed{\begin{array}{ll}
(c, e, s)^{\circ_n} & = c[\gamma \leftarrow e^{\circ_n} \uparrow^{\alpha}][\alpha \leftarrow s^{\circ_n}] \\
(c, e, s, v)^{\circ_n} & = c[\gamma \leftarrow e^{\circ_n} \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s^{\circ_n} \uparrow^{r}][r \leftarrow v^{\circ_n}] \\
(v \cdot e)^{\circ_n} & = v^{\circ_n} \cdot e^{\circ_n} \\
(v \cdot s)^{\circ_n} & = v^{\circ_n} \cdot s^{\circ_n} \\
(c[e])^{\circ_n} & = \mu\alpha.c[\gamma \leftarrow e^{\circ_n} \uparrow^{\alpha}] \\
\mathsf{tp}^{\circ_n} & = \mathsf{tp}
\end{array}}$$

Fig. 35. Translation of the call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine in $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$

is well-formed (written $(c, e, s) : \ \vdash \ )$ if r does not occur free in c. The well-formedness of the code is directly derived from the definition of the instructions. For example, the states

$$(\texttt{Extend-env}; c, e, s) \qquad \text{and} \qquad (\texttt{Lookup-env}; c, e, s, v)$$

are not well-formed, since one has:

$$(\texttt{Extend-env}; c) \ : \ (r \ \vdash \ \alpha, \gamma) \qquad \text{and} \qquad (\texttt{Lookup-env}; c) \ : \ (\ \vdash \ \alpha, \gamma)$$

Also, the closure $(\texttt{PushArg}; c)[e]$ is not well-formed since r occurs free in $\texttt{PushArg}; c$.

The machine starts with the empty environment and empty stack which are both represented with the top-level continuation. Its execution leads to well-formed configurations.

Proposition 5. *If st is well-formed and $st \to st'$ then st' is well-formed.*

Machine states are translated in the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus as shown in Figure 35. Starting from well-formed states the translation produces well-formed programs.

Proposition 6. *If st is a well-formed state of the call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine then st°_n} is a well-formed $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ program.*

The machine transitions correspond to the call-by-name evaluation strategy of the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$-calculus under the given translation.

Proposition 7. *Let st_1 and st_2 be well-formed states of the call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine. If $st_1 \to st_2$ then $(st_1{}^{\circ_n}) \xmapsto{cbn}\!\!\!\!\twoheadrightarrow st'_2$ such that $(st_2{}^{\circ_n})$ and st'_2 are equal up to permutation of substitutions.*

Proof. By cases on the transition. We only show three cases.

(`Exec`)

$$\begin{array}{rcl}
 & & (\texttt{Exec}, e, s, c[e']) \\
 & = & (\texttt{Exec})[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \\
 & = & \langle r \uparrow^{\alpha}\uparrow^{\gamma} \mid \alpha \uparrow^{r}\uparrow^{\gamma}\rangle[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \\
c\tau & \overset{cbn}{\longmapsto\!\!\!\!\rightarrow} & \langle r \uparrow^{\alpha}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid \\
 & & \qquad \alpha \uparrow^{r}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]]\rangle \\
e \uparrow^{k} 1 & \overset{cbn}{\longmapsto} & \langle r \uparrow^{\alpha}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid \\
 & & \alpha \uparrow^{r} [\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]]\rangle \\
e \uparrow^{r} 2 & \overset{cbn}{\longmapsto} & \langle r \uparrow^{\alpha}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid \\
 & & (\alpha[\alpha \leftarrow s]) \uparrow^{r} [r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]]\rangle \\
e \uparrow^{r} 1 & \overset{cbn}{\longmapsto} & \langle r \uparrow^{\alpha}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid (\alpha[\alpha \leftarrow s])\rangle \\
k\tau & \overset{cbn}{\longmapsto} & \langle r \uparrow^{\alpha}\uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid s\rangle \\
v \uparrow^{k} 1 & \overset{cbn}{\longmapsto} & \langle r \uparrow^{\alpha} [\alpha \leftarrow s \uparrow^{r}][r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid s\rangle \\
v \uparrow^{k} 1 & \overset{cbn}{\longmapsto} & \langle r[r \leftarrow \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}]] \mid s\rangle \\
r\tau & \overset{cbn}{\longmapsto} & \langle \mu\alpha.c[\gamma \leftarrow e' \uparrow^{\alpha}] \mid s\rangle \\
\mu & \overset{cbn}{\longmapsto} & c[\gamma \leftarrow e' \uparrow^{\alpha}][\alpha \leftarrow s]
\end{array}$$

Notice how after moving the substitutions inside the command, we start simplifying the consumer. After the consumer does not have any more $\mapsto_e$-redexes the producer is reduced. At the end, since s is not a $\tilde{\mu}$ term, the μ reduction is possible.

(`Closure`)

$$\begin{array}{rl}
 & (\mathsf{bind}\ \texttt{Closure}\ c\ \mathsf{in}\ c', e, s) \\
= & (\mathsf{bind}\ \texttt{Closure}\ c\ \mathsf{in}\ c')[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \\
= & \langle (\mu\alpha.c) \uparrow^{\alpha} \mid \tilde{\mu}r.c'\rangle[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \\
\overset{cbn}{\longmapsto\!\!\!\!\rightarrow} & \langle (\mu\alpha.c) \uparrow^{\alpha} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \mid (\tilde{\mu}r.c')[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s]\rangle \\
\overset{cbn}{\longmapsto\!\!\!\!\rightarrow} & \langle (\mu\alpha.c) \uparrow^{\alpha} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \mid \tilde{\mu}r.(c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}])\rangle \\
\overset{cbn}{\longmapsto} & \langle ((\mu\alpha.c)[\gamma \leftarrow e]) \uparrow^{\alpha} [\alpha \leftarrow s] \mid \tilde{\mu}r.(c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}])\rangle \\
\overset{cbn}{\longmapsto} & \langle (\mu\alpha.c)[\gamma \leftarrow e] \mid (\tilde{\mu}r.c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}])\rangle \\
\overset{cbn}{\longmapsto} & \langle \mu\alpha.(c[\gamma \leftarrow e \uparrow^{\alpha}]) \mid (\tilde{\mu}r.c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}])\rangle \\
\overset{cbn}{\longmapsto} & c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow (\mu\alpha.c[\gamma \leftarrow e \uparrow^{\alpha}])]
\end{array}$$

Notice how we resolve the critical pair in favor of the $\tilde{\mu}$-rule.

$$
\boxed{
\begin{array}{l}
\begin{array}{lcl}
st & ::= & \langle t, e, s\rangle \\
t & ::= & \bullet \mid \lambda t \mid (t\ t') \mid (t \uparrow) \\
e & ::= & v :: e \mid nil \\
s & ::= & v :: s \mid nil \\
v & ::= & [t, e]
\end{array} \\
\\
\begin{array}{lcl}
\langle \bullet, [t, e'] :: e, s\rangle & \rightarrow & \langle t, e', s\rangle \\
\langle t \uparrow, v :: e, s\rangle & \rightarrow & \langle t, e, s\rangle \\
\langle t\ u, e, s\rangle & \rightarrow & \langle t, e, [u, e] :: s\rangle \\
\langle \lambda t, e, v :: s\rangle & \rightarrow & \langle t, v :: e, s\rangle
\end{array}
\end{array}
}
$$

Fig. 36. The Krivine abstract machine

(`PopArg`)

$$
\begin{array}{rl}
 & (\texttt{PopArg};\ c, e, v \cdot s) \\
= & (\texttt{PopArg};\ c)[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow v \cdot s] \\
= & \langle (\lambda r.\mu\alpha.c) \uparrow^{\alpha}\ \mid \alpha \uparrow^{\gamma}\rangle[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow v \cdot s] \\
\xrightarrow{cbn}\!\!\!\!\rightarrow & \langle (\lambda r.\mu\alpha.c) \uparrow^{\alpha} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow v \cdot s] \mid \alpha \uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow v \cdot s]\rangle \\
\xrightarrow{cbn}\!\!\!\!\rightarrow & \langle (\lambda r.\mu\alpha.c) \uparrow^{\alpha} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow v \cdot s] \mid v \cdot s\rangle \\
\xmapsto{cbn} & \langle (\lambda r.\mu\alpha.c)[\gamma \leftarrow e] \uparrow^{\alpha} [\alpha \leftarrow (v \cdot s)] \mid v \cdot s\rangle \\
\xmapsto{cbn} & \langle (\lambda r.\mu\alpha.c)[\gamma \leftarrow e] \mid v \cdot s\rangle \\
\xmapsto{cbn} & \langle \lambda r.(\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}] \mid v \cdot s\rangle \\
\xmapsto{cbn} & \langle v \mid \tilde{\mu} r.\langle (\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}] \mid s \uparrow^{r}\rangle\rangle \\
\xmapsto{cbn} & \langle (\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}] \mid s \uparrow^{r}\rangle[r \leftarrow v] \\
\xrightarrow{cbn}\!\!\!\!\rightarrow & \langle (\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}][r \leftarrow v] \mid s \uparrow^{r} [r \leftarrow v]\rangle \\
\xmapsto{cbn} & \langle (\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}][r \leftarrow v] \mid s\rangle \\
\xrightarrow{cbn}\!\!\!\!\rightarrow & \langle \mu\alpha.c[\gamma \leftarrow e \uparrow^{r}\uparrow^{\alpha}][r \leftarrow v \uparrow^{\alpha}] \mid s\rangle \\
\xmapsto{cbn} & c[\gamma \leftarrow e \uparrow^{r}\uparrow^{\alpha}][r \leftarrow v \uparrow^{\alpha}][\alpha \leftarrow s]
\end{array}
$$

The above is equivalent to $c[\gamma \leftarrow e \uparrow^{r}\uparrow^{\alpha}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow v]$.

For multiple reduction steps the result follows since the order of the substitutions does not mask any potential redex. □

8.1.2 *The Krivine Machine.* The Krivine machine [2007] is a simple machine for implementing normal weak-head reduction of λ-terms. It has been shown correct by Wand [2007]. Its description is given in Figure 36. Our presentation of the machine is slightly different from the original presentation [Krivine 2007]: we restrict all λs to have one argument and we implement the environment lookup step-by-step. The state of the machine is represented by three components: the current λ_{DB}-term, the current environment, and the current argument stack. In contrast to the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ abstract machine there is no accumulator. The evaluation of $\bullet$ causes the evaluation of the closure on top of the stack. The code and environment saved in the closure become the current instructions and environment. The evaluation of $t \uparrow$ removes

$$\boxed{\begin{array}{lcl}
\langle t, e, s\rangle^{\bullet_n} & = & (\llbracket t\rrbracket, e^{\bullet_n}, s^{\bullet_n}) \\
nil^{\bullet_n} & = & \mathsf{tp} \\
(v :: e)^{\bullet_n} & = & v^{\bullet_n} \cdot e^{\bullet_n} \\
(v :: s)^{\bullet_n} & = & v^{\bullet_n} \cdot s^{\bullet_n} \\
([t, e])^{\bullet_n} & = & \llbracket t\rrbracket[e^{\bullet_n}]
\end{array}}$$

Fig. 37. Translating the Krivine machine into the call-by-name $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ abstract machine

the top element of the environment. To evaluate an application, the machine saves on the stack a closure made of the argument and the current environment, and proceeds with the term in function position. To evaluate a lambda-abstraction, the value (if any) on top of the stack is moved in the environment.

We translate the Krivine into the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ abstract machine, as shown in Figure 37. The translation faithfully represents the machine transitions.

Proposition 8. *Let st be a Krivine machine state.*

(i) $st^{\bullet_n}$ *is a well-formed state of the call-by-name* $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ *abstract machine;*
(ii) If $st \rightarrow st_1$ *then* $st^{\bullet_n} \twoheadrightarrow st_1{}^{\bullet_n}$.

We conclude that the Krivine machine transitions can be simulated by $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ call-by-name standard reduction.

Lemma 2. *The call-by-name semantics of the* $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$*-calculus is such that: if* $st \rightarrow st_1$ *in the Krivine machine, then* $st^{\bullet_n\circ_n} \xmapsto{cbn}\!\!\twoheadrightarrow st^{\bullet_n\circ_n}$.

Final states of the Krivine machine are of the form $\langle \lambda t, e, nil\rangle$. As shown in the following example they correspond to answers in the calculus modulo substitution.

Example 7. Let v be $\mu\alpha.\mathtt{Extend\text{-}env}; \llbracket t\rrbracket$. We have:

$$\begin{array}{lr}
\langle \lambda t, e, nil\rangle^{\bullet_n\circ_n} & = \\
\langle (\lambda r.v)\uparrow^{\alpha} \mid \alpha \uparrow^{\gamma}\rangle[\gamma \leftarrow e^{\bullet_n\circ_n}\uparrow^{\alpha}][\alpha \leftarrow \mathsf{tp}] & \twoheadrightarrow \\
\langle (\lambda r.v)\uparrow^{\alpha} [\gamma \leftarrow e^{\bullet_n\circ_n}\uparrow^{\alpha}][\alpha \leftarrow \mathsf{tp}] \mid \alpha \uparrow^{\gamma} [\gamma \leftarrow e^{\bullet_n\circ_n}\uparrow^{\alpha}][\alpha \leftarrow \mathsf{tp}]\rangle & \twoheadrightarrow \\
\langle (\lambda r.v)[\gamma \leftarrow e^{\bullet_n\circ_n}] \mid \mathsf{tp}\rangle & \rightarrow \\
\langle \lambda r.(v[\gamma \leftarrow e^{\bullet_n\circ_n}\uparrow^{r}]) \mid \mathsf{tp}\rangle &
\end{array}$$

8.2 Correspondence with the CEK Machine

The previous development applies with minor changes to the call-by-value case. We only present the main points. As before, we make use of an intermediate abstract machine. The call-by-value version of this intermediate machine is given in Figure 38. The environment is a sequence of closures. The stack does not only hold closures but also delayed contexts. Both closures and contexts are sequences of instructions paired with an environment. What distinguishes them is the tag: `arg` in case of a closure and `fun` in case of a context. We do not present the full definition of well-formed state which is similar to the one for the call-by-name machine. Note that the accumulator is free in a delayed context, since it is waiting for a value. The transition rules are similar to the call-by-name ones, with the exception of the `bind Closure` c `in` c', `PushArg` and `PopArg` instructions. The

Syntax:

$$\begin{array}{lll} \textit{state} & st & ::= (c, e, s) \mid (c, e, s, v) \\ \textit{environment} & e & ::= v \cdot e \mid \mathtt{tp} \\ \textit{stack} & s & ::= fr \cdot s \mid \mathtt{tp} \\ \textit{frame} & fr & ::= \mathtt{fun}(v) \mid \mathtt{arg}(v) \\ \textit{closure} & v & ::= c[e] \end{array}$$

Well-formed states:

$$\frac{c \ : \ (r \ \vdash \ \alpha, \gamma) \quad \mid e \ \vdash}{\vdash \ \mathtt{fun}(c[e])} \qquad \frac{c \ : \ (\ \vdash \ \alpha, \gamma) \quad \mid e \ \vdash}{\vdash \ \mathtt{arg}(c[e])}$$

Transition rules:

$$\begin{array}{lll} (\mathtt{Exec}, e, s, c[e']) & \to & (c, e', s) \\ (\mathtt{Clear}; c, e, s, v) & \to & (c, e, s) \\ (\mathsf{bind}\ \mathtt{Closure}\ c\ \mathsf{in}\ c', e, s) & \to & (c, e, \mathtt{fun}(c'[e]) \cdot s) \\ (\mathtt{PushArg}; c, e, s, v) & \to & (c, e, \mathtt{arg}(v) \cdot s) \\ (\mathtt{PopArg}; c, e, \mathtt{fun}(c'[e']) \cdot s) & \to & (c', e', s, (\mathtt{PopArg}; c)[e]) \\ (\mathtt{PopArg}; c, e, \mathtt{arg}(v) \cdot s) & \to & (c, e, s, v) \\ (\mathtt{Extend\text{-}env}; c, e, s, v) & \to & (c, v \cdot e, s) \\ (\mathtt{Lookup\text{-}env}; c, v \cdot e, s) & \to & (c, e, s, v) \end{array}$$

Fig. 38. The call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine

bind `Closure` c in c' instruction executes c and delays the execution of c' by making a delayed context and saving it on the stack. The `PushArg` instruction tags the accumulator's value and moves it on top of the stack. The `PopArg` instruction behaves differently depending on the top of the stack: if it is a closure, it removes the tag and moves the value in the accumulator, since it does not need any further evaluation; If it is a delayed context, then that context gets executed in the saved environment. The execution preserves well-formed states.

The translation of the call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine (written as $\cdot^{\circ_v}$) is given in Figure 39; given a well-formed state it returns a well-formed program. A result similar to Proposition 7 shows the correspondence of the call-by-value abstract machine and the call-by-value strategy of $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$.

PROPOSITION 9. *Let st_1 and st_2 be well-formed states of the call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine. If $st_1 \to st_2$ then $(st_1{}^{\circ_v}) \xmapsto{cbv}\!\!\!\!\!\twoheadrightarrow st_2'$ such that $(st_2{}^{\circ_v})$ and st_2' are equal up to permutation of substitutions.*

PROOF. In the proof of Proposition 7, notice how all the steps in the simulation of the `Exec` instruction are also valid for the call-by-value strategy. The same is true for the `PopArg` instruction when the stack contains a closure, and for the `Extend-env` and `Lookup-env` instructions. Instead, the bind `Closure` c in c' instruction depends

$$\begin{array}{|lcl|}\hline
(c,e,s)^{\circ_v} & = & c[\gamma \leftarrow e^{\circ_v} \uparrow^{\alpha}][\alpha \leftarrow s^{\circ_v}] \\
(c,e,s,v)^{\circ_v} & = & c[\gamma \leftarrow e^{\circ_v} \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s^{\circ_v} \uparrow^{r}][r \leftarrow v^{\circ_v}] \\
(v \cdot e)^{\circ_v} & = & v^{\circ_v} \cdot e^{\circ_v} \\
(\mathtt{fun}(c[e]) \cdot s)^{\circ_v} & = & \tilde{\mu} r.c[\gamma \leftarrow e^{\circ_v} \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s^{\circ_v} \uparrow^{r}] \\
(\mathtt{arg}(v) \cdot s)^{\circ_v} & = & v^{\circ_v} \cdot s^{\circ_v} \\
((\mathtt{PopArg}; c)[e])^{\circ_v} & = & \lambda r.(\mu\alpha.c)[\gamma \leftarrow e^{\circ_v} \uparrow^{r}] \\
(c[e])^{\circ_v} & = & \mu\alpha.c[\gamma \leftarrow e^{\circ_v} \uparrow^{\alpha}] \\
\mathtt{tp}^{\circ_v} & = & \mathtt{tp} \\ \hline
\end{array}$$

Fig. 39. Translation of the call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine in $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$

on the strategy:

$$\begin{array}{cl}
 & (\mathsf{bind}\ \mathtt{Closure}\ c\ \mathsf{in}\ c', e, s)^{\circ_v} \\
= & (\mathsf{bind}\ \mathtt{Closure}\ c\ \mathsf{in}\ c')[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \\
= & \langle (\mu\alpha.c) \uparrow^{\alpha} \mid \tilde{\mu} r.c' \rangle[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow s] \\
\overset{cbv}{\longmapsto\!\!\!\!\rightarrow} & \langle \mu\alpha.(c[\gamma \leftarrow e \uparrow^{\alpha}]) \mid \tilde{\mu} r.(c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}]) \rangle \\
\overset{cbv}{\longmapsto} & c[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow \tilde{\mu} r.c'[\gamma \leftarrow e \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}]]
\end{array}$$

Notice how in the last step the $\tilde{\mu}$ is given priority.

The simulation of the other case of the PopArg instruction is as follows:

$$\begin{array}{cl}
 & (\mathtt{PopArg};\ c, e, \mathtt{fun}(c'[e']) \cdot s)^{\circ_v} \\
= & (\mathtt{PopArg};\ c)[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow \tilde{\mu} r.c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}]] \\
= & \langle (\lambda r.\mu\alpha.c) \uparrow^{\alpha} \mid \alpha \uparrow^{\gamma} \rangle[\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow \tilde{\mu} r.c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}]] \\
\overset{cbv}{\longmapsto\!\!\!\!\rightarrow} & \langle (\lambda r.\mu\alpha.c) \uparrow^{\alpha} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow \tilde{\mu} r.c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}]] \mid \\
 & \qquad \alpha \uparrow^{\gamma} [\gamma \leftarrow e \uparrow^{\alpha}][\alpha \leftarrow \tilde{\mu} r.c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}]] \rangle \\
\overset{cbv}{\longmapsto\!\!\!\!\rightarrow} & \langle \lambda r.(\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}] \mid \tilde{\mu} r.c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}] \rangle \\
\overset{cbv}{\longmapsto\!\!\!\!\rightarrow} & c'[\gamma \leftarrow e' \uparrow^{\alpha}\uparrow^{r}][\alpha \leftarrow s \uparrow^{r}][r \leftarrow \lambda r.(\mu\alpha.c)[\gamma \leftarrow e \uparrow^{r}]]
\end{array}$$

□

The call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine corresponds to the right-to-left CEK machine given in Figure 40. The state of the machine is described by two kinds of states: $\langle t, e, s \rangle$ and $\langle v, s \rangle$. As before, the environment is a list of closures, and the stack is a list of two types of delayed contexts: $(t\ \square)[e]$ indicates a computation waiting for the argument to be computed, and $(\square\ v)$ indicates a computation waiting for the function to be applied.

The embedding (written as $\cdot^{\bullet_v}$) of a CEK machine state is given in Figure 41. As for the Krivine machine, final states are of the form $\langle \lambda t, e, nil \rangle$. As shown in Example 7, they correspond to answers modulo substitution.

Proposition 10. *Let st and st_1 be states of the right-to-left CEK machine. If $st \rightarrow st_1$ then $st^{\bullet_v} \twoheadrightarrow st_1{}^{\bullet_v}$.*

$$\begin{array}{lll}
st & ::= & \langle t, e, s\rangle \mid \langle v, s\rangle \\
v & ::= & t[e] \\
e & ::= & v :: e \mid nil \\
f & ::= & (t\ \square)[e] \mid (\square\ v) \\
s & ::= & f :: s \mid nil
\end{array}$$

$$\begin{array}{lll}
\langle \bullet, v :: e, s\rangle & \rightarrow & \langle v, s\rangle \\
\langle t \uparrow, v :: e, s\rangle & \rightarrow & \langle t, e, s\rangle \\
\langle \lambda t, e, s\rangle & \rightarrow & \langle (\lambda t)[e], s\rangle \\
\langle t_1\ t_2, e, s\rangle & \rightarrow & \langle t_2, e, (t_1\ \square)[e] :: s\rangle \\
\langle (\lambda t)[e], (t_1\ \square)[e'] :: s\rangle & \rightarrow & \langle t_1, e', (\square\ (\lambda t)[e]) :: s\rangle \\
\langle (\lambda t)[e], (\square\ v) :: s\rangle & \rightarrow & \langle t, v :: e, s\rangle
\end{array}$$

Fig. 40. Right-to-left CEK abstract machine

$$\begin{array}{lll}
\langle t, e, s\rangle^{\bullet_v} & = & (\llbracket t\rrbracket, e^{\bullet_v}, s^{\bullet_v}) \\
\langle t[e], s\rangle^{\bullet_v} & = & (\llbracket t\rrbracket, e^{\bullet_v}, s^{\bullet_v}) \\
nil^{\bullet_v} & = & \mathtt{tp} \\
(v :: e)^{\bullet_v} & = & v^{\bullet_v} \cdot e^{\bullet_v} \\
t[e]^{\bullet_v} & = & \llbracket t\rrbracket[e^{\bullet_v}] \\
(f :: s)^{\bullet_v} & = & f^{\bullet_v} \cdot s^{\bullet_v} \\
((t\ \square)[e])^{\bullet_v} & = & \mathtt{fun}((\mathtt{PushArg}; \llbracket t\rrbracket)[e^{\bullet_v}]) \\
(\square\ v)^{\bullet_v} & = & \mathtt{arg}(v^{\bullet_v})
\end{array}$$

Fig. 41. Translation of the right-to-left CEK abstract machine into the call-by-value $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$ abstract machine

Lemma 3. *The call-by-value semantics of the $\lambda\mu\tilde{\mu}\mathtt{r}\!\uparrow$-calculus is such that: if $st \rightarrow st'$ in the right-to-left CEK machine, then $st^{\bullet_v \circ_v} \xmapsto{cbv}\!\!\!\!\rightarrow st'^{\bullet_v \circ_v}$.*

9. TYPE PRESERVATION

Compilation is essentially a proof transformation [Ohori 2005]. It embeds the proof system of Figure 11 into the one of Figure 42, as shown next.

Proposition 11. *Given a well-typed λ_{DB} term t, $\llbracket t\rrbracket$ is well-typed. Moreover, if $A_1, \cdots, A_n \vdash t : B$ and T is an atomic type then*

$$\llbracket t\rrbracket : (\vdash \alpha : B, \gamma : A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)$$

Proof. The proof is by rule induction and proceeds by cases:

(*Axiom*) Let us assume $A_1, \cdots, A_n, B \ \vdash\ \bullet : B$. We have:

$$\cfrac{\overline{\mathtt{Exec}\ :\ (r : B\ \vdash\ \alpha : B, \gamma : A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)}}{\mathtt{Lookup\text{-}env};\ \mathtt{Exec}\ :\ (\ \vdash\ \alpha : B, \gamma : B \rightarrow A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)}$$

$$\frac{}{\texttt{Exec} \ : \ (r : A \ \vdash \ \alpha : A, \gamma : B)}$$

$$\frac{c \ : \ (\ \vdash \ \alpha : B, \gamma : C)}{(\texttt{Clear}; \ c) \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C)}$$

$$\frac{c \ : \ (\ \vdash \ \alpha : A \rightarrow B, \gamma : C)}{(\texttt{PushArg}; \ c) \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C)}$$

$$\frac{c \ : \ (\ \vdash \ \gamma : A \rightarrow B, \alpha : C)}{(\texttt{Extend-env}; \ c) \ : \ (r : A \ \vdash \ \gamma : B, \alpha : C)}$$

$$\frac{c \ : \ (\ \vdash \ \alpha : A, \gamma : D) \quad c' \ : \ (r : A \ \vdash \ \alpha : C, \gamma : D)}{(\text{bind } \texttt{Closure} \ c \text{ in } c') \ : \ (\ \vdash \ \alpha : C, \gamma : D)}$$

$$\frac{c \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C)}{(\texttt{PopArg}; \ c) \ : \ (\ \vdash \ \alpha : A \rightarrow B, \gamma : C)}$$

$$\frac{c \ : \ (r : A \ \vdash \ \gamma : B, \alpha : C)}{(\texttt{Lookup-env}; \ c) \ : \ (\ \vdash \ \gamma : A \rightarrow B, \alpha : C)}$$

Fig. 42. Typing the instructions

(Weakening) Let us assume $A_1, \cdots, A_n, B \ \vdash \ t\uparrow : C$.

$$\frac{\dfrac{[\![t]\!] \ : \ (\ \vdash \ \alpha : C, \gamma : A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)}{\texttt{Clear}; \ [\![t]\!] \ : \ (r : B \ \vdash \ \alpha : C, \gamma : A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)}}{\texttt{Lookup-env}; \ \texttt{Clear}; \ [\![t]\!] \ : \ (\ \vdash \ \alpha : C, \gamma : B \rightarrow A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T)}$$

($\rightarrow_e$) Let us assume $A_1, \cdots, A_n \ \vdash \ t_1 \ t_2 : B$. We let C stands for $A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T'$. We have:

$$\frac{[\![t_2]\!] \ : \ (\ \vdash \ \alpha : A, \gamma : C) \quad \dfrac{[\![t_1]\!] \ : \ (\ \vdash \ \alpha : A \rightarrow B, \gamma : C)}{\texttt{PushArg}; \ [\![t_1]\!] \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C)}}{\text{bind } \texttt{Closure} \ [\![t_2]\!] \text{ in } \texttt{PushArg}; \ [\![t_1]\!] \ : \ (\ \vdash \ \alpha : B, \gamma : C)}$$

($\rightarrow_i$) Let us assume $A_1, \cdots, A_n \ \vdash \ \lambda t : A \rightarrow B$. We let C stands for $A_n \rightarrow \cdots \rightarrow A_1 \rightarrow T'$.

$$\frac{\dfrac{[\![t]\!] \ : \ (\ \vdash \ \alpha : B, \gamma : A \rightarrow C)}{\texttt{Extend-env}; \ [\![t]\!] \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C)}}{\texttt{PopArg}; \ \texttt{Extend-env}; \ [\![t]\!] \ : \ (\ \vdash \ \alpha : A \rightarrow B, \gamma : C)}$$

□

A compiler error can be captured as a type error. For example, if one erroneously compiles the term $\lambda\bullet$ as:

$$\texttt{PopArg}; \texttt{Extend-env}; \texttt{PopArg}; \texttt{Extend-env}; \texttt{Lookup-env}; \texttt{Exec}$$

an error is raised since it is not possible to derive the following judgment:

$$\begin{array}{l}\texttt{PopArg}; \texttt{Extend-env}; \texttt{PopArg}; \texttt{Extend-env}; \texttt{Lookup-env}; \texttt{Exec} \ : \\ (\ \vdash\ \alpha : A \rightarrow A, \gamma : C)\end{array}$$

One can define the notion of well-formed instruction by removing the type information from Figure 42. Even this notion can help in finding compiler errors. For example, if one compiles the λ_{DB} term $((\lambda\bullet)\ t_1)$ as

$$\textsf{bind}\ \texttt{Closure}\ [\![t_1]\!]\ \textsf{in}\ \texttt{PopArg}; \texttt{Extend-env}; \texttt{Lookup-env}; \texttt{Exec}$$

one would not be able to derive the judgement:

$$\textsf{bind}\ \texttt{Closure}\ [\![t_1]\!]\ \textsf{in}\ \texttt{PopArg}; \texttt{Extend-env}; \texttt{Lookup-env}; \texttt{Exec}\ :\ (\ \vdash\ \alpha, \gamma)$$

since the following is not derivable:

$$\texttt{PopArg}; \texttt{Extend-env}; \texttt{Lookup-env}; \texttt{Exec}\ :\ (r\ \vdash\ \alpha, \gamma)$$

Also correctness of optimizations can be based on proofs transformations. For example, the compilation of a known call:

$$[\![(\lambda t)t']\!] = \textsf{bind}\ \texttt{Closure}\ [\![t']\!]\ \textsf{in}\ \texttt{PushArg}; \texttt{PopArg}; \texttt{Extend-env}; [\![t]\!]$$

can be optimized as follows:

$$\textsf{bind}\ \texttt{Closure}\ [\![t']\!]\ \textsf{in}\ \texttt{Extend-env}; [\![t]\!]$$

This optimization corresponds to an elimination of a detour. The proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\![t]\!]\ :\ (\ \vdash\ \alpha : \beta, \gamma : A \rightarrow C)}{\texttt{Extend-env}; [\![t]\!]\ :\ (r : A\ \vdash\ \alpha : B, \gamma : C)}}{\texttt{PopArg}; \texttt{Extend-env}; [\![t]\!]\ :\ (\ \vdash\ \alpha : A \rightarrow B, \gamma : C)}}{\texttt{PushArg}; \texttt{PopArg}; \texttt{Extend-env}; [\![t]\!]\ :\ (r : A\ \vdash\ \alpha : B, \gamma : C)} \quad [\![t']\!]\ :\ (\ \vdash\ \alpha : A, \gamma : C)}{\textsf{bind}\ \texttt{Closure}\ [\![t']\!]\ \textsf{in}\ \texttt{PushArg}; \texttt{PopArg}; \texttt{Extend-env}; [\![t]\!]\ :\ (\ \vdash\ \alpha : B, \gamma : C)}$$

is transformed to:

$$\dfrac{\dfrac{[\![t]\!]\ :\ (\ \vdash\ \alpha : \beta, \gamma : A \rightarrow C)}{\texttt{Extend-env}; [\![t]\!]\ :\ (r : A\ \vdash\ \alpha : B, \gamma : C)} \quad [\![t']\!]\ :\ (\ \vdash\ \alpha : A, \gamma : C)}{\textsf{bind}\ \texttt{Closure}\ [\![t']\!]\ \textsf{in}\ \texttt{Extend-env}; [\![t]\!]\ :\ (\ \vdash\ \alpha : B, \gamma : C)}$$

Next, we define a type system for the Krivine and the CEK machines and show that well-typed states translate to well-typed commands. The typing of the Krivine machine is given in Figure 43. It makes use of the following judgements:

$$\Gamma\ \vdash\ t : A \qquad st\ :\ (\ \vdash\ ) \qquad |\ s : A\ \vdash \qquad |\ e : A\ \vdash \qquad \vdash\ v : A\ |$$

The first judgement is defined in Figure 11. A Krivine state $\langle t, e, s\rangle$ is well-typed if the stack's type corresponds to the type of t, the environment's type corresponds to the types of the assumptions used in typing t. T stands for an atomic type, this guarantees that if n is the number of assumptions then e must contain n elements. For example,

$$\langle \bullet, nil, nil\rangle \qquad \langle (\bullet \uparrow) \uparrow, v :: nil, nil\rangle$$

are not well-typed. Also, if $\vdash\ \lambda\bullet : A \rightarrow A$ and $\vdash\ v : B$ then

$$\langle \lambda\bullet, nil, v :: nil\rangle$$

$$\frac{A_1,\cdots,A_n \vdash t:C \quad | \, s:C \vdash \quad | \, e:A_n \to \cdots \to A_1 \to T \vdash}{\langle t,e,s\rangle \ : \ (\ \vdash\ )}$$

$$\frac{}{| \, nil : A \vdash}$$

$$\frac{\vdash v:A \, | \quad | \, s:B \vdash}{| \, v :: s : A \to B \vdash} \qquad \frac{\vdash v:A \, | \quad | \, e:B \vdash}{| \, v :: e : A \to B \vdash}$$

$$\frac{A_1,\cdots,A_n \vdash t:C \, | \quad | \, e:A_n \to \cdots \to A_1 \to T \vdash}{\vdash [t,e]:C \, |}$$

Fig. 43. Type system for the Krivine machine

$$\frac{A_1,\cdots,A_n \vdash t:C \quad | \, s:C \vdash \quad | \, e:A_n \to \cdots \to A_1 \to T \vdash}{\langle t,e,s\rangle \ : \ (\ \vdash\ )}$$

$$\frac{\vdash v:A \, | \quad | \, s:A \vdash}{\langle v,s\rangle \ : \ (\ \vdash\ )}$$

$$\frac{}{| \, nil : A \vdash}$$

$$\frac{A_1,\cdots,A_n \vdash t:A \to C \, | \quad | \, s:C \vdash \quad | \, e:A_n \to \cdots \to A_1 \to T \vdash}{| \, (t\ \Box)\ [e] :: s : A \vdash}$$

$$\frac{\vdash v:A \, | \quad | \, s:B \vdash}{| \, (\Box\ v) :: s : A \to B \vdash}$$

$$\frac{\vdash v:A \, | \quad | \, e:B \vdash}{| \, v :: e : A \to B \vdash}$$

$$\frac{A_1,\cdots,A_n \vdash t:C \, | \quad | \, e:A_n \to \cdots \to A_1 \to T \vdash}{\vdash t[e]:C \, |}$$

Fig. 44. Type system for the CEK machine

is not well-typed. The Krivine machine satisfies subject reduction.

PROPOSITION 12. *Given Krivine states s and s', if $s\ :\ (\ \vdash\ )$ and $s \to s'$ then $s'\ :\ (\ \vdash\ )$.*

LEMMA 4. *Given a Krivine state s. If s is well-typed then $s^{\bullet_n \circ_n}$ is well-typed.*

PROOF. It follows from the above proposition and by rule induction. □

$$\frac{c \ : \ (r : A \ \vdash \ \alpha : B, \gamma : C) \quad \mid e : C \ \vdash \quad \mid s : B \ \vdash \quad \vdash \ v : A \mid}{(c, e, s, v) : \ \vdash}$$

$$\frac{c \ : \ (\ \vdash \ \alpha : B, \gamma : C) \quad \mid e : C \ \vdash \quad \mid s : B \ \vdash}{(c, e, s) : \ \vdash}$$

$$\frac{}{\mid \mathsf{tp} : A \ \vdash}$$

$$\frac{\vdash \ v : A \mid \quad \mid e : B \ \vdash}{\mid v \cdot e : A \to B \ \vdash} \qquad \frac{\vdash \ v : A \mid \quad \mid s : B \ \vdash}{\mid v \cdot s : A \to B \ \vdash}$$

$$\frac{c \ : \ (\ \vdash \ \alpha : B, \gamma : C) \quad \mid e : C \ \vdash}{\vdash \ c[e] : B \mid}$$

Fig. 45. Type system for the call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine

The typing for the CEK machine is given in Figure 44. The CEK machine satisfies subject reduction.

Proposition 13. *Given CEK states s and s', if $s \ : \ (\ \vdash \ )$ and $s \to s'$ then $s' \ : \ (\ \vdash \ )$.*

Lemma 5. *Given a CEK state s. If s is well-typed then $s^{\bullet_v \circ_v}$ is well-typed.*

Proof. It follows from Proposition 11 and by rule induction. □

The typing for the $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machines are easily derived from Figures 34 and 38. For example, the typing rules for the call-by-name $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ abstract machine are given in Figure 45.

10. CONCLUSIONS AND FUTURE WORK

Natural deduction is usually taken as the logical foundation of λ-calculus. We have shown that the sequent calculus offers a better correspondence with abstract machines. This translates into the ability to define a call-by-name and a call-by-value semantics in a *non-recursive manner*, unlike the corresponding semantics defined in a natural deduction setting. This means that the next redex to be executed always occurs at a bounded distance from the root of the syntax tree. In other words, the semantics defines a *tail-recursive* evaluator. To demonstrate that our semantics lead to an answer we have shown they correspond to standard reductions of a call-by-name and call-by-value $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ calculi which are defined by orienting a critical pair. Moreover, we have shown that Krivine and the right-to-left CEK abstract machines are implementations of these standard reductions.

This work constitutes our first step towards providing a Curry-Howard isomorphism for low-level code. Next, we plan to build on $\lambda\mu\tilde{\mu}\mathbf{r}\!\uparrow$ to create a suitable logic for embedding "real" machines. We will consider the JVM, and the different levels of the TAL abstract machines [Morrisett et al. 2002]. We would like to explain the certifying compilation as a transformation among different proof systems.

By interpreting the machine code as a term in a suitable logic, we also envision an alternative approach of proof-carrying code based on program extraction. Instead of sending a program and a proof of its safety, the program and the proof are sent together in a single proof-term. In other words, the program itself is a proof. This is all one needs for both type checking the proof and recovering the underlying program. The safety of execution is guaranteed by the subject reduction property. The correctness of the entire approach is guaranteed by the embedding of the non-standard logics into well established logics.

We also plan to make use of these kinds of logics, which naturally embed run-time data-structures such as the notions of control stack and environment, in the formalization of analysis such as stack inspection [Fournet and Gordon 2002] and access control [Pottier et al. 2001]. These formalizations will be *inside* the logic itself rather than on *top* of it. Thus permitting a better understanding of the interaction between these analysis, optimizations and alternative reduction strategies.

REFERENCES

Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. 1990. Explicit substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 31–46.

Ager, M. S., Biernacki, D., Danvy, O., and Midtgaard, J. 2003. A functional correspondence between evaluators and abstract machines. Research Series RS-03-13, BRICS. March. 26 pp.

Appel, A. 2001. Foundational proof-carrying code. In *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif. `http://www.cs.princeton.edu/~appel/papers/fpcc.pdf`.

Barthe, G., Dufay, G., Jakubiec, L., Serpette, B. P., and de Sousa, S. M. 2001. A formal executable semantics of the JavaCard platform. In *Proceedings of the European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, 302–319.

Curien, P.-L., Hardin, T., and Lévy, J.-J. 1996. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM 43,* 2, 362–397.

Curien, P.-L. and Herbelin, H. 2000. The duality of computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, 233–243.

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

Douence, R. and Fradet, P. 1998. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems 20,* 2 (March), 344–387.

Felleisen, M., Friedman, D., and Kohlbecker, E. 1987. A syntactic theory of sequential control. *Theoretical Computer Science 52(3)*, 205–237.

Felleisen, M., Friedman, D., Kohlbecker, E., and Duba, B. 1986. Reasoning with continuations. In *First Symposium on Logic and Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 131–141.

Felleisen, M. and Friedman, D. P. 1986. Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Language Concepts III*. Elsevier, Amsterdam, The Netherlands, 193–217.

Fournet, C. and Gordon, A. D. 2002. Stack inspection: theory and variants. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 307–318. `http://research.microsoft.com/~fournet/papers/stack-inspection-theory-and-variants-popl-02.pdf`.

Gentzen, G. 1969. Investigations into logical deduction. In *Collected Papers of Gerhard Gentzen*, M. Szabo, Ed. Elsevier, Amsterdam, The Netherlands, 68–131.

GRIFFIN, T. G. 1990. The formulae-as-types notion of control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 47–57.

HARDIN, T., MARANGET, L., AND PAGANO, B. 1996. Functional back-ends within the lambda-sigma calculus. In *International Conference on Functional Programming*. ACM Press, New York, 25–33.

HERBELIN, H. 1994. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Annual Conference of the European Association for Computer Science Logic, Kazimierz, Poland*. Lecture Notes in Computer Science, vol. 933. Springer-Verlag, Berlin.

HIGUCHI, T. AND OHORI, A. 2002. Java bytecode as a typed term calculus. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, New York, 201–211. `http://www.jaist.ac.jp/~ohori/research/jvmcalc.ps`.

HOWARD, W. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, J. R. Hindley and J. P. Seldin, Eds. Elsevier, Amsterdam, The Netherlands, 479–490.

HUET, G. AND LÉVY, J.-J. 1991. Computations in orthogonal rewriting systems, i. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, MA, 395–414.

JONES, M. P. 1998. The functions of Java bytecode. In *Proceedings of the OOPSLA Wokshop on the Formal Underpinnings of Java*. Imperial College of Science, Technology, and Medicine, London.

KLEIN, G. AND STRECKER, M. 2004. Verified bytecode verification and type-certifying compilation. *Journal of Logic Programming 58,* 1-2, 27–60. citeseer.ist.psu.edu/article/klein03verified.html.

KRIVINE, J.-L. 2007. A call-by-name lambda calculus machine. To appear in Higher Order and Symbolic Computation.

LESCANNE, P. 1994. From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 60–69.

LIU, H. AND MOORE, J. S. 2004. Java program verification via a JVM deep embedding in ACL2. In *17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*. Lecture Notes in Computer Science, vol. 3223. Springer-Verlag, Berlin, 184–200.

MORRISETT, J. G., CRARY, K., GLEW, N., AND WALKER, D. 2002. Stack-based typed assembly language. In *Journal of Functional Programming*. Cambridge University Press, Cambridge, Mass. `http://www.cs.cornell.edu/talc/papers/stal-tic-abstract.html`.

OHORI, A. 2005. A proof theory for machine code. Available from `http://www.pllab.riec.tohoku.ac.jp/~ohori/research/LogicalMachineRevOct2005.pdf`.

PARIGOT, M. 1993. Classical proofs as programs. *Computational Logic and Theory 713*, 263–276.

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

POLONOVSKI, E. 2004. Strong normalization of $\overline{\lambda}\mu\tilde{\mu}$-calculus. In *Foundations of Software Science and Computation Structures (FOSSACS 2004)*. Lecture Notes in Computer Science, vol. 2987. Springer-Verlag, Berlin, 423–437.

POTTIER, F., SKALKA, C., AND SMITH, S. F. 2001. A systematic approach to static access control. In *Proceedings of the European Symposium on Programming Languages and Systems*. Springer-Verlag, Berlin, 30–45. `http://www.cs.uvm.edu/~skalka/skalka-pubs/fpottier-skalka-smith-toplas03.pdf`.

PRAWITZ, D. 1965. *Natural Deduction, a Proof-Theoretical Study*. Almquist and Wiksell, Stockholm.

REUS, B. AND STREICHER, T. 1998. Classical logic, continuation semantics and abstract machines. *J. Funct. Prog. 8,* 6, 543–572.

WADLER, P. 2003. Call-by-value is dual to call-by-name. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York.

WAND, M. 2007. On the correctness of the Krivine machine. To appear in Higher Order and Symbolic Computation Special Issue on the Krivine Machine.

Yelland, P. M. 1999. A compositional account of the Java virtual machine. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 57–69.