AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

2015 Workshop on Continuations

Department of Computer Science

Olivier Danvy

Pre-proceedings

2015 Workshop on Continuations
– pre-proceedings –

Olivier Danvy

London, UK, 12 April 2015

This volume contains the papers presented at WoC 2015, the Workshop on Continuations held at ETAPS 2015.

There were four submissions. Each of them was reviewed by, on the average, three PC members. The committee decided to accept three papers. The program also includes one invited talk. It also documents the depth, variety, and richness of continuations with four distilled tutorials.

Thanks are due to the local organizers of ETAPS 2015 for the infrastructure and to the general chairman of WoC 2015, Ugo de'Liguoro, for initiating this workshop and making it happen.

Program of the 2015 Workshop on Continuations

09:00 - 10:00
  Invited Talk:
    "A verified abstract machine for functional coroutines"
    Tristan Crolard, CNAM, France

10:00 - 10:30
  Contributed Talk:
    "The selection monad as a CPS translation"
    Jules Hedges, Queen Mary University of London, UK

11:00 - 11:30
  Contributed Talk:
    "A modular structural operational semantics for delimited continuations"
    Neil Sculthorpe, Paolo Torrini, and Peter Mosses, Swansea University, UK

11:30 - 12:30
  Distilled Tutorial:
    "Why all programmers want continuations (but use callbacks instead)"
    Gabriel Kerneis, Paris, France

14:00 - 15:00
  Distilled Tutorial:
    "Command injection attacks, continuations, and the Lambek calculus"
    Hayo Thielecke, University of Birmingham, UK

15:00 - 16:00
  Distilled Tutorial:
    "Bisimulations for delimited-control operators"
    Dariusz Biernacki, University of Wroclaw, Poland
    Serguei Lenglet, Université de Lorraine, France

16:30 - 17:00
  Contributed Talk:
    "ATM without tears:
     prompt-passing style transformation for typed delimited-control operators"
    Ikuo Kobori and Yukiyoshi Kameyama, University of Tsukuba, Japan
    Oleg Kiselyov, Tohoku University, Japan

17:00 - 18:00
  Distilled Tutorial:
    "Logical by need"
    Alexis Saurin, CNRS & Université Paris Diderot -- Paris 7, France
    Pierre-Marie Pédrot, Université Paris Diderot -- Paris 7, France

A verified abstract machine for functional coroutines

Tristan Crolard
CNAM, France
tristan.crolard@cnam.fr

March 4, 2015

Abstract

Functional coroutines are a restricted form of control mechanism, where each continuation comes with its local environment. This restriction was originally obtained by considering a constructive version of Parigot's classical natural deduction which is sound and complete for the Constant Domain logic. In this article, we present a refinement of de Groote's abstract machine which is proved to be correct for functional coroutines. Therefore, this abstract machine also provides a direct computational interpretation of the Constant Domain logic.

Contents

1 Introduction

The *Constant Domain* logic (CD) is a well-known intermediate logic due to Grzegorczyk [25] which can be characterized as a logic for Kripke frames with constant domains. Although CD is semantically simpler than intuitionistic logic, its proof theory is quite difficult : no *conventional* cut-free axiomatization is known [31], and it took more than three decades to prove that the interpolation theorem does not hold either [33]. However, CD is unavoidable when the object of study is *duality in intuitionistic logic*. Indeed, consider the following schema (called either D [25] or DIS [37]), where x does not occur free in B:

$$\forall x(A \vee B) \vdash (\forall x A) \vee B$$

The dual of this schema is $(\exists x A) \wedge B \vdash \exists x(A \wedge B)$ which is clearly valid in intuitionistic logic. Thus bi-intuitionistic logic (also called Heyting-Brouwer logic [37] or subtractive logic [11]), which contains both intuitionistic logic and dual intuitionistic logic, includes both schemas.

Görnemann proved that the addition of the DIS-schema to intuitionistic predicate logic is sufficient to axiomatize CD [23] (and also that the disjunction and existence properties hold, so CD is still a constructive logic). Moreover, Rauszer proved that bi-intuitionistic is conservative over CD [37]. As a consequence, we should expect at least the same difficulties with the proof-theoretical study of bi-intuitionistic logic as with CD. In particular, if we want to understand the computational content of bi-intuitionistic logic, it is certainly worth spending some time on CD.

Although there is no conventional cut-free axiomatization of CD, there are some non-conventional deduction systems which do enjoy cut elimination. The first such system was defined by Kashima [27] as a restriction of Gentzen's sequent calculus LK based on dependency relations. Independently, we described [9] a similar restriction using Parigot's classical natural deduction [35] instead of LK. Another difference lies in the fact that our restriction can also be formulated at the level of proof-terms (terms of the $\lambda\mu$-calculus in Parigot's system), independently of the typing derivation. Such proof-terms, which are terms of Parigot's $\lambda\mu$-calculus, are called *safe* in our calculus [12]. The intuition behind this terminology is presented informally in the introduction of this article as follows:

> "[...] we observe that in the restricted $\lambda\mu$-calculus, even if continuations are no longer first-class objects, the ability of context-switching remains (in fact, this observation is easier to make in the framework of abstract state machines). However, a context is now a pair $\langle$*environment*, *continuation*$\rangle$. Note that such a pair is exactly what we expect as the context of a coroutine, since a coroutine should not access the local environment (the part of the environment which is not shared) of another coroutine. Consequently, we say that a $\lambda\mu$-term t is *safe with respect to coroutine contexts* (or

just *safe* for short) if no coroutines of t access the local environment of another coroutine."

In this paper, we provide some evidence to support this claim in the framework of abstract state machines. As a starting point, we take an environment machine for the $\lambda\mu$-calculus, which is defined and proved correct by de Groote [18] (a very similar machine was defined independently by Streicher and Reus [40]). Then we define a new variant of this machine dedicated to the execution of safe terms (which works exactly as hinted above). Note that this modified machine is surprisingly simpler than what we would expect form the negative proof-theoretic results. We actually obtain a direct, meaningful, computational interpretation of the Constant Domain logic, even though dependency relations were at the beginning only a complex technical device.

As usual with environment machines, it was more convenient to encode variables as de Bruijn indices. Since safe $\lambda\mu$-terms have different scoping rules than regular $\lambda\mu$-terms, the translation into de Bruijn terms should yield different terms: safe $\lambda\mu$-terms need to use *local indices* to access the local environment of the current coroutine, whereas regular terms use the usual *global indices* to access the usual global environment.

As a consequence of this remark, we obtain a proof of correctness of the modified machine which is two-fold. We first introduce an intermediate machine *with local indices, global environment and indirection tables*, then we define two functional bi-simulations $(-)^\star$ and $(-)^\diamond$ showing that this intermediate machine:

- bi-simulates the regular machine with global indices
- bi-simulates the modified machine with local environments

The combined bi-simulation then show that the modified machine is correct with respect to de Groote's regular machine:

$$\begin{array}{lccccccccc}
\text{Regular machine} & \tilde{\sigma}_0^\star & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n^\star & \rightsquigarrow & \tilde{\sigma}_{n+1}^\star & \rightsquigarrow & \cdots \\
 & \uparrow \star & & & & \uparrow \star & & \uparrow \star & & \\
\text{Intermediate machine} & \tilde{\sigma}_0 & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n & \rightsquigarrow & \tilde{\sigma}_{n+1} & \rightsquigarrow & \cdots \\
 & \downarrow \diamond & & & & \downarrow \diamond & & \downarrow \diamond & & \\
\text{Modified machine} & \tilde{\sigma}_0^\diamond & \rightsquigarrow & \cdots & \rightsquigarrow & \tilde{\sigma}_n^\diamond & \rightsquigarrow & \tilde{\sigma}_{n+1}^\diamond & \rightsquigarrow & \cdots
\end{array}$$

The plan of the paper is the following. In Section 2, we first recall the notion of safety [9], and then we present a simpler (but equivalent) definition of safety which is more convenient for the proofs of correctness. In Section 3, we present the regular and the modified machine. Finally, in Section 4, we detail the proof of correctness: we describe the intermediate machine and the two functional bi-simulations (all the proofs were mechanically checked with the Coq proof assistant).

1.1 Related works

Computational interpretation of classical logic

Since Griffin's pioneering work [24], the extension of the well-known formulas-as-types paradigm to classical logic has been widely investigated for instance by Murthy [34], Barbanera and Berardi [2], Rehof and Sørensen [38], de Groote [19], and Krivine [30]. We shall consider here Parigot's $\lambda\mu$-calculus mainly because it is confluent and strongly normalizing in the second order framework [35]. Note that Parigot's original CND is a second-order logic, in which $\vee, \wedge, \exists, \exists^2$ are definable from $\rightarrow, \forall, \forall^2$. An extension of CND with primitive conjunction and disjunction has also been investigated by Pym, Ritter and Wallen [36] and de Groote [19].

The computational interpretation of classical logic is usually given by a λ-calculus extended with some form of control (such as the famous **call/cc** of Scheme or the catch/throw mechanism of Lisp) or similar formulations of first-class continuation constructs. Continuations are used in denotational semantics to describe control commands such as jumps. They can also be used as a programming technique to simulate backtracking and coroutines. For instance, first-class continuations have been successfully used to implement Simula-like cooperative coroutines in Scheme [22]. This approach has been extended in the Standard ML of New Jersey (with the typed counterpart of Scheme's **call/cc** [26]) to provide simple and elegant implementations of light-weight processes (or threads), where concurrency is obtained by having individual threads voluntarily suspend themselves [39]. The key point in these implementations is that control operators make it possible to switch between coroutine contexts, where the context of a coroutine is encoded as its continuation.

Coroutines

The concept of coroutine is usually attributed to Conway [8] who introduced it to describe the interaction between a lexer and a parser inside a compiler. They are also used by Knuth [29] which sees them as a mechanism that generalizes subroutines (procedures without parameters). Coroutines first appeared in the language Simula-67 [15]. A formal framework for proving the correctness of simple programs containing coroutines has also been developed [7]. Coroutine mechanisms were later introduced in several programming language, for instance in Modula-2 [42], and more recently in the functional language Lua [21].

In his thesis, Marlin [32] summarizes the characteristics of a coroutine as follows:

1. *the values of data local to a coroutine persist between successive occasions on which control enters it (that is, between successive calls), and*
2. *the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.*

That is, a coroutine is a subroutine *with a local state* which can suspend and resume execution. This informal definition is of course not sufficient to capture

the various implementations that have been developed in practice. To be more specific, the main differences between coroutine mechanisms can be summarized in as follows [20]:

- *the control-transfer mechanism, which can provide symmetric or asymmetric coroutines.*
- *whether coroutines are provided in the language as first-class objects, which can be freely manipulated by the programmer, or as constrained constructs;*
- *whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from with nested calls.*

Symmetric coroutines generally offer a single control-transfer operation that allows coroutines to pass control between them. Asymmetric control mechanisms, sometimes called *semi-coroutines* [14], rely on two primitives for the transfer of control: the first to invoke a coroutine, the second to pause and return control to the caller.

A well-known illustration of the third point above, called "the same-fringe problem", is to determine whether two trees have exactly the same sequence of leaves using two coroutines, where each coroutine recursively traverses a tree and passes control to other coroutine when it encounters a leaf. The elegance of this algorithm lies in the fact that each coroutine uses its own stack, which permits for a simple recursive tree traversal. Note however that there are also other solutions of this problem which do not rely on coroutines [5].

Asymmetric coroutines often correspond to the coroutines mechanism made directly accessible to the programmer (as in Simula or Lua), sometimes as a restricted form of *generators* (as in C#). On the other hand, symmetric coroutines are generally chosen as a low-level mechanism used to implement more advanced concurrency mechanisms (as in Modula). An other example is the Unix Standard [41] where the recommended low-level primitives for implementing lightweight processes (users threads) are *getcontext*, *setcontext*, *swapcontext* and *makecontext*. This is the terminology we have previously adopted for our coroutines [12]. However, since we are working in a purely functional framework, we shall write "functional coroutines" to avoid any confusion with other mechanisms.

More recently, Anton and Thiemann described a static type system for first-class, stackful coroutines [1] that may be used in both, symmetric and asymmetric ways. They followed Danvy's method [16] to derive definitional interpreters for several styles of coroutines from the literature (starting from reduction semantics). This work is clearly very close to our formalization, and it should help shed some light on these mechanisms. However, we should keep in mind that logical deduction systems come with their own constraints which might not be compatible with existing programming paradigms.

$$x : \Gamma, A^x \vdash \Delta; A$$

$$\frac{t : \Gamma, A^x \vdash \Delta; B}{\lambda x.t : \Gamma \vdash \Delta; A \to B}(I_\to) \qquad \frac{t : \Gamma \vdash \Delta; A \to B \qquad u : \Gamma \vdash \Delta; A}{t\ u : \Gamma \vdash \Delta; B}(E_\to)$$

$$\frac{t : \Gamma \vdash \Delta; A}{\mathbf{throw}\ \alpha\ t : \Gamma \vdash \Delta, A^\alpha; B}(W_R) \qquad \frac{t : \Gamma \vdash \Delta, A^\alpha; A}{\mathbf{catch}\ \alpha\ t : \Gamma \vdash \Delta; A}(C_R)$$

Table 1: Classical Natural Deduction

2 Dependency relations

Parigot's original CND is a deduction system for the second-order classical logic. Since we are mainly interested here in the computational content of untyped terms, we shall simply recall the restriction in the propositional framework corresponding to classical logic with the implication as only connective (in Table 1). We refer the reader to [9, 12] for the full treatment of primitive conjunction, disjunction and quantifiers (including the proof that the restricted system is sound and complete for CD).

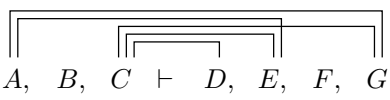
Remark. We actually work with a minor variant of the original $\lambda\mu$-calculus, called the λ_{ct}-calculus, with a primitive catch/throw mechanism [10]. These primitives are however easily definable in the $\lambda\mu$-calculus as $\mathbf{catch}\ \alpha\ t \equiv \mu\alpha[\alpha]t$ and $\mathbf{throw}\ \alpha\ t \equiv \mu\delta[\alpha]t$ where δ is a name which does not occur in t.

Since Parigot's CND is multiple-conclusioned sequent calculus, it is possible to apply so-called *Dragalin restriction* to obtain a sound and complete system for CD. This restriction requires that the succedent of the premise of the introduction rule for implication have only one formula:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash \Delta, A \to B}$$

Unfortunately, the Dragalin restriction is not stable under proof-reduction. However, a weaker restriction which is stable under proof-reduction, consists in allowing multiple conclusions in the premise of this rule, with the proviso that *these other conclusions do not depend on* A. These dependencies between occurrences of hypotheses and occurrences of conclusions in a sequent are defined by induction on the derivation.

Example. Consider a derived sequent $A, B, C \vdash D, E, F, G$ with the following dependencies:



Using named hypotheses $A^x, B^y, C^z \vdash D, E, F, G$, this annotated sequent may be represented as:

$$A^x, B^y, C^z \vdash \{z\} : D, \{x, z\} : E, \{\} : F, \{x, z\} : G$$

Let us assume now that t is the proof-term corresponding to the above derivation, i.e. we have derived in CND the following typing judgment:

$$t : A^x, B^y, C^z \vdash D^\alpha, E^\beta, F^\gamma; G$$

then we could also obtain the same dependencies directly from t, by computing sets of variables used by the various coroutines (where $[]$ refers to the distinguished conclusion), and we would get:

- $\mathcal{S}_\alpha(t) = \{z\}$
- $\mathcal{S}_\beta(t) = \{x, z\}$
- $\mathcal{S}_\gamma(t) = \{\}$
- $\mathcal{S}_{[]}(t) = \{x, z\}$

There is thus no need to actually annotate sequents with dependency relations: the relevant information is already present inside the proof-term. Let us recall how these sets are defined [12].

Definition 1. *Given a term t, for any free μ-variable δ of t, the sets of variables $\mathcal{S}_\delta(v)$ and $\mathcal{S}_{[]}(u)$ are defined inductively as follows:*

- $\mathcal{S}_{[]}(x) = \{x\}$
  $\mathcal{S}_\delta(x) = \emptyset$
- $\mathcal{S}_{[]}(\lambda x.u) = \mathcal{S}_{[]}(u) \backslash \{x\}$
  $\mathcal{S}_\delta(\lambda x.u) = \mathcal{S}_\delta(u) \backslash \{x\}$
- $\mathcal{S}_{[]}(u\ v) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_{[]}(v)$
  $\mathcal{S}_\delta(u\ v) = \mathcal{S}_\delta(u) \cup \mathcal{S}_\delta(v)$
- $\mathcal{S}_{[]}(\mathbf{catch}\ \alpha\ u) = \mathcal{S}_{[]}(u) \cup \mathcal{S}_\alpha(u)$
  $\mathcal{S}_\delta(\mathbf{catch}\ \alpha\ u) = \mathcal{S}_\delta(u)$
- $\mathcal{S}_{[]}(\mathbf{throw}\ \alpha\ u) = \emptyset$
  $\mathcal{S}_\alpha(\mathbf{throw}\ \alpha\ u) = \mathcal{S}_\alpha(u) \cup \mathcal{S}_{[]}(u)$
  $\mathcal{S}_\delta(\mathbf{throw}\ \alpha\ u) = \mathcal{S}_\delta(u)$ *for any* $\delta \neq \alpha$

Definition 2. *A term t is* **safe** *if and only if for any subterm of t which has the form $\lambda x.u$, for any free μ-variable δ of u, $x \notin \mathcal{S}_\delta(u)$.*

Example. The term $\lambda x.\mathbf{catch}\ \alpha\ \lambda y.\mathbf{throw}\ \alpha\ x$ is safe, since x was declared before **catch** α and x is thus visible in **throw** $\alpha\ x$. On the other hand, $\lambda x.\mathbf{catch}\ \alpha\ \lambda y.\mathbf{throw}\ \alpha\ y$ is not safe, because y is not visible in **throw** $\alpha\ y$. More generally, for any α, a term of the form $\lambda y.\mathbf{throw}\ \alpha\ y$ is the reification of α as a first-class continuation and such a term is never safe. This can also be understood at the type level since the typing judgment of such a term is the law of excluded middle $\vdash A^{\alpha}; \neg A$.

Remark. You can thus decide *a posteriori* if a proof in CND is valid in CD simply by checking if the (untyped) proof-term is safe.

2.1 Safety revisited

In the conventional λ-calculus, there are two standard algorithms to decide whether a term is closed: either you build inductively the set of free variables and then check that it is empty, or you define a recursive function which takes as argument the set of declared variables, and check that each variable has been declared.

Similarly, for the $\lambda\mu$-calculus there are two ways of defining safety: the previous definition refined the standard notion of free variable (by defining a set per free μ-variable). In the following definition, *Safe* takes as arguments the sets of visible variables for each coroutine, and then decides for each variable, if the variable is visible in the current coroutine. For a closed term, *Safe* is called with $\mathcal{V}, \mathcal{V}_\mu$ both empty.

Definition 3. *The property* $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$ *by induction on* t *as follows:*

$$\begin{array}{rcl}
Safe^{\mathcal{V},\mathcal{V}_\mu}(x) & = & x \in \mathcal{V} \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(t\ u) & = & Safe^{\mathcal{V},\mathcal{V}_\mu}(t) \wedge Safe^{\mathcal{V},\mathcal{V}_\mu}(u) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\lambda x.t) & = & Safe^{(x::\mathcal{V}),\mathcal{V}_\mu}(t) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\mathbf{catch}\ \alpha\ t) & = & Safe^{\mathcal{V},(\alpha\mapsto\mathcal{V};\mathcal{V}_\mu)}(t) \\
Safe^{\mathcal{V},\mathcal{V}_\mu}(\mathbf{throw}\ \alpha\ t) & = & Safe^{\mathcal{V}_\mu(\alpha),\mathcal{V}_\mu}(t)
\end{array}$$

where:

- $\mathcal{V}$ *is a list of variables*
- $\mathcal{V}_\mu$ *maps* μ*-variables onto lists of variables*

Remark. This definition can also be seen as the reformulation, at the level of proof-terms, of the "top-down" definition of the restriction of CND from [6] which was introduced in the framework of proof search.

As expected, we can show that the above two definitions of safety are equivalent. More precisely, the following propositions are provable.

Proposition 4. *For any term t and any mapping $\mathcal{V}_\mu$ such that $FV_\mu(t) \subseteq dom(\mathcal{V}_\mu)$, we have: $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$ implies $\mathcal{S}_{[]}(t) \subseteq \mathcal{V}$ and $\mathcal{S}_\delta(t) \subseteq \mathcal{V}_\mu(\delta)$ for any $\delta \in dom(\mathcal{V}_\mu)$ and t is safe.*

Proposition 5. *For any safe term t, for any set $\mathcal{V}$ such that $\mathcal{S}_{[]}(t) \subseteq \mathcal{V}$, for any mapping $\mathcal{V}_\mu$ such that $FV_\mu(t) \subseteq dom(\mathcal{V}_\mu)$ and $\mathcal{S}_\delta(t) \subseteq \mathcal{V}_\mu(\delta)$ for any $\delta \in FV_\mu(t)$, we have $Safe^{\mathcal{V},\mathcal{V}_\mu}(t)$.*

3 Abstract machines

In this section, we recall Groote's abstract machine for the $\lambda\mu$-calculus [18], then we present the modified machine for safe terms and we prove its correctness. But before we describe the abstract machines, we need to move to a syntax using *de Bruijn* indices, and to adapt the definition of safety.

3.1 Safe $\lambda_{\mathbf{ct}}$-terms

We rely on *de Bruijn* indices for both kind of variables (the regular variables and the μ-variables) but they correspond to different name spaces. Let us now call *vector* a list of indices (i.e. natural numbers), and *table* a list of vectors. The definition of *Safe* given for named terms can be rephrased for *de Bruijn* terms as follows. For a closed term, *Safe* is called with $\mathcal{I}$, $\mathcal{I}_\mu$ both empty and $n = 0$.

Notation 6. **We write $\backquote{g}\backquote$ for the term corresponding to variable with index g. The rest of the syntax is standard for de Bruijn terms.*

Definition 7. *Given t: term, $\mathcal{I}$: vector, $\mathcal{I}_\mu$: table and n: nat, the property $Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t)$ is defined inductively by the following rules:*

$$\frac{n - g = k \qquad k \in \mathcal{I}}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\backquote{g}\backquote)}$$

$$\frac{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t) \qquad Safe_n^{\mathcal{I},\mathcal{I}_\mu}(u)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t\ u)}$$

$$\frac{Safe_{Sn}^{(Sn::\mathcal{I}),\mathcal{I}_\mu}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\lambda t)}$$

$$\frac{Safe_n^{\mathcal{I},(\mathcal{I}::\mathcal{I}_\mu)}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\mathbf{catch}\ t)}$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad Safe_n^{\mathcal{I}',\mathcal{I}_\mu}(t)}{Safe_n^{\mathcal{I},\mathcal{I}_\mu}(\mathbf{throw}\ \alpha\ t)}$$

Remark. Note that n is used to count occurrences of λ from the root of the term (seen as a tree), and such a number clearly uniquely determines a λ on a branch. Since they are the numbers stored in $\mathcal{I}$, $\mathcal{I}_\mu$, a difference is computed for the base case since *de Bruijn* indices count λ beginning with the leaf.

3.2 From local indices to global indices

In the framework of environment machines, the point of using *de Bruijn* indices to represent variables is to point directly to the location of the closure in the environment (the closure which is bound to the variable). On the other hand, the intuition behind the safety property is that for each continuation, there is only a fragment of the environment which is visible (the local environment of the coroutine).

In the modified machine, these indices should point to locations in the local environment. Although the abstract syntaxes are isomorphic, it seems better to introduce a new calculus (since indices in terms have different semantics), where we can also rename **catch/throw** as **get-context/set-context** (to be consistent with the new semantics). Let us call $\lambda_{\mathbf{gs}}$-calculus the resulting calculus, and let us now define formally the translation of $\lambda_{\mathbf{gs}}$-terms onto (safe) $\lambda_{\mathbf{ct}}$-terms.

Remark. In the Coq proof assistant, it is often more convenient to represent partial functions as relations (since all functions are total in Coq we would need option types to encode partial functions). In the sequel, we call "functional" or "deterministic" any relation which has been proved functional.

Definition 8. *The functional relation* $\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t) = t'$*, with* t*:* $\lambda_{\mathbf{gs}}$*-term,* t'*:* $\lambda_{\mathbf{ct}}$*-term,* $\mathcal{I}$*: vector,* $\mathcal{I}_\mu$*: table and* n*: nat, is defined inductively by the following rules:*

$$\frac{n - \mathcal{I}(l) = g}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\text{`}l\text{`}) = (\text{`}g\text{`})}$$

$$\frac{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t) = t' \qquad \downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (u) = u'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (t\ u) = (t'\ u')}$$

$$\frac{\downarrow_{Sn}^{(Sn::\mathcal{I}),\mathcal{I}_\mu} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\lambda t) = (\lambda t')}$$

$$\frac{\downarrow_n^{\mathcal{I},(\mathcal{I}::\mathcal{I}_\mu)} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\mathbf{get\text{-}context}\ t) = (\mathbf{catch}\ t')}$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad \downarrow_n^{\mathcal{I}',\mathcal{I}_\mu} (t) = t'}{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu} (\mathbf{set\text{-}context}\ \alpha\ t) = (\mathbf{throw}\ \alpha\ t')}$$

The shape of this definition is obviously very similar to the definition of safety. Actually, we can prove that a $\lambda_{\mathbf{ct}}$-term is safe if and only if it is the image of some $\lambda_{\mathbf{gs}}$-term by the translation.

Lemma 9. $\forall\ \mathcal{I}\ \mathcal{I}_\mu\ n\ t'$, $Safe_n^{\mathcal{I},\mathcal{I}_\mu}(t') \quad \leftrightarrow \quad \exists t, \downarrow_n^{\mathcal{I},\mathcal{I}_\mu}(t) = t'$.

3.3 Abstract machine for $\lambda_{\mathbf{ct}}$-terms

De Groote's machine [18] is actually an extension of the well-known Krivine's abstract machine which has already been studied extensively in the literature [17]. Moreover, this abstract machine can also be mechanically derived from a contextual semantics of the $\lambda\mu$-calculus with explicit substitutions using the method developed by Biernacka and Danvy [4], and it is thus correct by construction.

3.3.1 Closure, environment, stack and state

Definition 10. *A closure is an inductively defined tuple* $[t, \mathcal{E}, \mathcal{E}_\mu]$ *with* t *: term,* $\mathcal{E}$ *: closure list,* $\mathcal{E}_\mu$ *: stack list (where a stack is a closure list).*

Definition 11. *A state is defined as a tuple* $\langle\ t,\ \mathcal{E},\ \mathcal{E}_\mu,\ \mathcal{S}\ \rangle$ *where* $[t, \mathcal{E}, \mathcal{E}_\mu]$ *is a closure and* $\mathcal{S}$ *is a stack.*

3.3.2 Evaluation rules

Definition 12. *The deterministic transition relation* $\sigma_1 \rightsquigarrow \sigma_2$*, with* σ_1*,* σ_2*: state, is defined inductively by the following rules:*

$$\frac{\mathcal{E}(k) = [t, \mathcal{E}', \mathcal{E}'_\mu]}{\langle \text{`}k\text{`}, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu)\,, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, \mathcal{E}_\mu, [u, \mathcal{E}, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, \mathcal{E}, \mathcal{E}_\mu, c :: \mathcal{S}\rangle \rightsquigarrow \langle t, (c :: \mathcal{E}), \mathcal{E}_\mu, \mathcal{S}\rangle$$

$$\langle \mathbf{catch}\ t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \mathbf{throw}\ \alpha\ t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

3.4 Abstract machine for $\lambda_{\mathbf{gs}}$-terms (with local environments)

As mentioned in the introduction, the modified abstract machine for λ_{gs}-terms is a surprisingly simple variant of de Groote's abstract machine, where a μ-variable is mapped onto a pair $\langle$*environment*, *continuation*$\rangle$ (a context) and not only a continuation. For simplicity, we still keep two distinct environments in the machine, $\mathcal{L}_\mu$ and $\mathcal{E}_\mu$ in a closure (but they have the same domain, which is the set of free μ-variables). As expected, the primitives **get-context** and **set-context** respectively capture and restore the local environment together with the continuation.

3.4.1 Closure, environment, stack and state

Definition 13. *A closure$_l$ is an inductively defined tuple $[t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu]$ where t: term, $\mathcal{L}$: closure$_l$ list, and $\mathcal{L}_\mu$: (closure$_l$ list) list, $\mathcal{E}_\mu$: stack$_l$ list (where a stack$_l$ is closure$_l$ list).*

Definition 14. *A state$_l$ is defined as a tuple $\langle\, t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S} \,\rangle$ where $[t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu]$ is a closure$_l$ and $\mathcal{S}$ is a stack$_l$.*

3.4.2 Evaluation rules

Definition 15. *The deterministic transition relation $\sigma_1 \rightsquigarrow^l \sigma_2$, with σ_1, σ_2: state$_l$, is defined inductively by the following rules:*

$$\frac{\mathcal{L}(k) = [t, \mathcal{L}', \mathcal{L}'_\mu, \mathcal{E}'_\mu]}{\langle \text{`}k\text{'}, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}', \mathcal{L}'_\mu, \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu)\,, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, [u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, c :: \mathcal{S}'\rangle \rightsquigarrow^l \langle t, (c :: \mathcal{L}), \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}'\rangle$$

$$\langle \textbf{get-context}\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, (\mathcal{L} :: \mathcal{L}_\mu), (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{L}_\mu(\alpha) = \mathcal{L}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}', \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

4 Bisimulations

We first introduce the intermediate machine for λ_{gs}-terms, then we define two functional bi-simulations $(-)^\star$ and $(-)^\diamond$ showing that this intermediate machine:

- bi-simulates the regular machine with global indices
- bi-simulates the modified machine with local environments

4.1 Abstract machine for λ_{gs}-terms (with indirection tables)

This intermediate machine for λ_{gs}-terms works *with local indices, global environment and indirection tables*. The indirection tables are exactly the same as for the static translation of λ_{gs}-terms to safe λ_{ct}-terms. However, the translation is now performed at runtime. The lock-step simulation $(-)^\star$ shows that translating during evaluation is indeed equivalent to evaluating the translated term. The lock-step simulation $(-)^\diamond$ shows that we can "flatten away" the indirection tables and the global environment, and work only with local environments.

4.1.1 Closure, environment, stack and state

Definition 16. *A closure$_i$ is an inductively defined tuple $[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]$, with t : term, n : nat, $\mathcal{I}$: vector, $\mathcal{I}_\mu$: table, $\mathcal{E}$: closure$_i$ list, $\mathcal{E}_\mu$: stack$_i$ list (where a stack$_i$ is a closure$_i$ list).*

Definition 17. *A state$_i$ is defined as a tuple $\langle\ t,\ n,\ \mathcal{I},\ \mathcal{I}_\mu,\ \mathcal{E},\ \mathcal{E}_\mu,\ \mathcal{S}\ \rangle$ where $[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]$ is a closure$_i$ and $\mathcal{S}$ is a stack$_i$.*

4.1.2 Evaluation rules

Definition 18. *The deterministic transition relation $\sigma_1 \rightsquigarrow^i \sigma_2$, with σ_1, σ_2: state$_i$, is defined inductively by the following rules:*

$$\frac{n - \mathcal{I}(l) = g \qquad \mathcal{E}(g) = [t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu]}{\langle \backslash l \backslash, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

$$\langle (tu), n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, [u, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

$$\langle \lambda t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, c :: \mathcal{S}'\rangle \rightsquigarrow^i \langle t, (Sn), ((Sn) :: \mathcal{I}), \mathcal{I}_\mu, c :: \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle$$

$$\langle \textbf{get-context}\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, (\mathcal{I} :: \mathcal{I}_\mu), \mathcal{E}, (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}', \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

4.2 Lock-step simulation $(-)^\star$

Definition 19. *The functional relation $c^\star =_c c'$, with c: closure$_i$, c': closure, is defined by the following rule:*

$$\frac{\downarrow_n^{\mathcal{I},\mathcal{I}_\mu}(t) = u \qquad \mathcal{E}^\star =_e \mathcal{E}' \qquad \mathcal{E}^\star_\mu =_k \mathcal{E}'_\mu}{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu]}$$

where $\mathcal{E}^\star$ and $\mathcal{E}^\star_\mu$ are defined by element-wise application of *.

Definition 20. *The functional relation $\sigma^\star =_\sigma \sigma'$, with σ: state$_i$, σ': state, is defined by the following rule:*

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu] \qquad \mathcal{S}^\star =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\star =_\sigma \langle u, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}'\rangle}$$

where $\mathcal{S}^\star$ is defined by element-wise application of *.

4.2.1 Soundness

Theorem 21. $\forall\ \sigma_1\ \sigma_2\ \sigma_1',$
$\sigma_1 \rightsquigarrow^i \sigma_2\ \rightarrow\ \sigma_1^\star =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2',\ \sigma_1' \rightsquigarrow \sigma_2'\ \wedge\ \sigma_2^\star =_\sigma \sigma_2'.$

4.2.2 Completeness

Theorem 22. $\forall\ \sigma_1'\ \sigma_2'\ \sigma_1,$
$\sigma_1' \rightsquigarrow \sigma_2'\ \rightarrow\ \sigma_1^\star =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2,\ \sigma_1 \rightsquigarrow^i \sigma_2\ \wedge\ \sigma_2^\star =_\sigma \sigma_2'.$

4.3 Lock-step simulation $(-)^\diamond$

Definition 23. *The functional relation $c^\diamond =_k c'$ with c: closure$_i$, c': closure$_l$, is defined by the following rule:*

$$\frac{\mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I}\ =\ \mathcal{L} \qquad \mathit{map}\ (\mathit{flatten}\ n\ \mathcal{E})\ \mathcal{I}_\mu\ =\ \mathcal{L}_\mu \qquad \mathcal{E}_\mu^\diamond =_k \mathcal{E}_\mu'}{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\diamond =_c [t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu']}$$

where $\mathcal{S}^\diamond$ and $\mathcal{E}_\mu^\diamond$ are defined by element-wise application of $^\diamond$, and the functional relation ($\mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I} = \mathcal{L}$) is defined inductively by the following rules:

$$\mathit{flatten}\ n\ \mathcal{E}\ \mathit{nil}\ =\ \mathit{nil} \qquad \frac{\mathcal{E}(n-k) = c \qquad c^\diamond =_c c' \qquad \mathit{flatten}\ n\ \mathcal{E}\ \mathcal{I}\ =\ \mathcal{L}}{\mathit{flatten}\ n\ \mathcal{E}\ (k :: \mathcal{I})\ =\ (c' :: \mathcal{L})}$$

Definition 24. *The functional relation $\sigma^\diamond =_\sigma \sigma'$, with σ : state$_i$, σ': state$_l$, is defined by the following rule:*

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\diamond =_c [u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu'] \qquad \mathcal{S}^\diamond =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\diamond =_\sigma \langle u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu', \mathcal{S}'\rangle}$$

4.3.1 Soundness

Theorem 25. $\forall\ \sigma_1\ \sigma_2\ \sigma_1',$
$\sigma_1 \rightsquigarrow^i \sigma_2\ \rightarrow\ \sigma_1^\diamond =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2',\ \sigma_1' \rightsquigarrow^l \sigma_2'\ \wedge\ \sigma_2^\diamond =_\sigma \sigma_2'.$

4.3.2 Completeness

Theorem 26. $\forall\ \sigma_1'\ \sigma_2'\ \sigma_1,$
$\sigma_1' \rightsquigarrow^l \sigma_2'\ \rightarrow\ \sigma_1^\diamond =_\sigma \sigma_1'\ \rightarrow\ \exists \sigma_2,\ \sigma_1 \rightsquigarrow^i \sigma_2\ \wedge\ \sigma_2^\diamond =_\sigma \sigma_2'.$

5 Conclusion and future work

We have defined and formally proved the correctness of an abstract machine which provides a direct computational interpretation of the Constant Domain logic. However, as mentioned in the introduction, this work is a stepping stone towards a computational interpretation of duality in intuitionistic logic. Starting from the reduction semantics of proof-terms of bi-intuitionistic logic (subtractive

logic) [12], it should be possible to extend the modified machine to account for first-class coroutines.

These results should then be compared with other related works, such as Curien and Herbelin's pioneering article on the duality of computation [13], or more recently, Bellin and Menti's work on the π-calculus and co-intuitionistic logic [3] and Kimura and Tatsuta's Dual Calculus [28].

Acknowledgments. I would like to thank Nuria Brede for numerous discussions on the "safe" $\lambda\mu$-calculus and useful comments on earlier versions of this work. I am also very grateful to Olivier Danvy for proof-reading this article, and for giving me the opportunity to present these results at WoC 2015.

References

[1] K. Anton and P. Thiemann. Towards deriving type systems and implementations for coroutines. In Kazunori Ueda, editor, *Programming Languages and Systems – 8th Asian Symposium, APLAS 2010*, volume 6461 of *LNCS*, pages 63–79, Shanghai, China, 2010. Springer. 4

[2] F. Barbanera and S. Berardi. Extracting Constructive Content from Classical Logic via Control-like Reductions. In *LNCS*, volume 662, pages 47–59. Springer-Verlag, 1994. 3

[3] G. Bellin and A. Menti. On the π-calculus and Co-intuitionistic Logic. Notes on Logic for Concurrency and λP Systems. *Fundamenta Informaticae*, 130(1):21–65, January 2014. 14

[4] M. Biernacka and O. Danvy. A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines. *Theoretical Computer Science*, 375, 2007. 10

[5] D. Biernacki, O. Danvy, and C. Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006. 4

[6] N. Brede. $\lambda\mu$PRL - A Proof Refinement Calculus for Classical Reasoning in Computational Type Theory. Master's thesis, University of Potsdam, 2009. 7

[7] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2(1):50–63, 1973. 3

[8] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. 3

[9] T. Crolard. Extension de l'Isomorphisme de Curry-Howard au Traitement des Exceptions (application d'une étude de la dualité en logique intuitionniste). Thèse de Doctorat. Université Paris 7, 1996. 1, 2, 5

[10] T. Crolard. A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming*, 9(6):625–647, 1999. 5

[11] T. Crolard. Subtractive Logic. *Theoretical Computer Science*, 254(1–2):151–185, 2001. 1

[12] T. Crolard. A Formulæ-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation*, 14(4):529–570, 2004. 1, 4, 5, 6, 14

[13] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 233–243, New York, USA, 2000. ACM Press. 14

[14] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press, 1972. 4

[15] O.-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966. 3

[16] O. Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 131–142, New York, USA, 2008. ACM Press. 4

[17] O. Danvy, editor. Special Issue on the Krivine Machine. *Higher-Order and Symbolic Computation*, 20(3), 2007. 10

[18] P. de Groote. An environment machine for the lambda-mu-calculus. *Mathematical Structure in Computer Science*, 8:637–669, 1998. 2, 8, 10

[19] P. de Groote. Strong normalization of classical natural deduction with disjunction. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *LNCS*, pages 182–196. Springer, 2001. 3

[20] A. L. de Moura and R. Ierusalimschy. Revisiting Coroutines. MCC 15/04, PUC-Rio, Rio de Janeiro, RJ, June 2004. 4

[21] A. L. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004. 3

[22] D. P. Friedman, C. T. Haynes, and M. Wand. Obtaining Coroutines with Continuations. *Journal of Computer Languages*, 11(3/4):143–153, 1986. 3

[23] S. Görnemann. A logic stronger than intuitionism. *The Journal of Symbolic Logic*, 36:249–261, 1971. 1

[24] T. G. Griffin. A formulæ-as-types notion of control. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Langages*, pages 47–58, 1990. 3

[25] A. Grzegorczyk. A philosophically plausible formal interpretation of intuitionistic logic. *Nederl. Akad. Wet., Proc., Ser. A*, 67:596–601, 1964. 1

[26] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993. 3

[27] R. Kashima. Cut-Elimination for the intermediate logic CD. Research Report on Information Sciences C100, Institute of Technology, Tokyo, 1991. 1

[28] D. Kimura and M. Tatsuta. Dual calculus with inductive and coinductive types. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil*, volume 5595 of *LNCS*, pages 224–238. Springer, 2009. 14

[29] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 2nd edition edition, 1973. 3

[30] J.-L. Krivine. Classical logic, storage operators and second order λ-calculus. *Ann. of Pure and Appl. Logic*, 68:53–78, 1994. 3

[31] E. G. K. Lopez-Escobar. A Second Paper "On the Interpolation Theorem for the Logic of Constant Domains". *The Journal of Symbolic Logic*, 48(3):595–599, 1983. 1

[32] C. D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1980. 3

[33] G. Mints, G. Olkhovikov, and A. Urquhart. Failure of interpolation in constant domain intuitionistic logic. *The Journal of Symbolic Logic*, 78:937–950, 9 2013. 1

[34] C. R. Murthy. Classical proofs as programs: How, when, and why. Technical Report 91-1215, Cornell University, Department of Computer Science, 1991. 3

[35] M. Parigot. Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*, 1993. 1, 3

[36] D. Pym, E. Ritter, and L. Wallen. On the intuitionistic force of classical search. *Theoretical Computer Science*, 232(1-2):299–333, 2000. 3

[37] C. Rauszer. An algebraic and Kripke-style approach to a certain extension of intuitionistic logic. In *Dissertationes Mathematicae*, volume 167. Institut Mathématique de l'Académie Polonaise des Sciences, 1980. 1

[38] N. J. Rehof and M. H. Sørensen. The λ_Δ-calculus. In *Theoretical Aspects of Computer Software*, volume 542 of *LNCS*, pages 516–542. Springer-Verlag, 1994. 3

[39] J. H. Reppy. First-class synchronous operations. In *Proceedings of the International Workshop on Theory and Practice of Parallel Programming*, volume 907 of *LNCS*, pages 235–252, London, UK, 1995. Springer-Verlag. 3

[40] T. Streicher and B. Reus. Classical Logic, Continuation Semantics and Abstract Machines. *Journal of Functional Programming*, 8(6):543–572, 1998. 2

[41] The Open Group. The Single UNIX Specification, Version 2, 1997. 4

[42] N. Wirth and J. Mincer-Daszkiewicz. *Modula-2*. ETH Zurich, Schweiz, 1980. 3

The selection monad as a CPS translation

Jules Hedges

February 28, 2015

Abstract

A computation in the continuation monad returns a final result given a continuation, ie. it is a function with type $(X \to R) \to R$. If we instead return the intermediate result at X then our computation is called a selection function. Selection functions appear in diverse areas of mathematics and computer science (especially game theory, proof theory and topology) but the existing literature does not heavily emphasise the fact that the selection monad is a CPS translation. In particular it has so far gone unnoticed that the selection monad has a call/cc-like operator with interesting similarities and differences to the usual call/cc, which we explore experimentally using Haskell.

Selection functions can be used whenever we find the intermediate result more interesting than the final result. For example a SAT solver computes an assignment to a boolean function, and then its continuation decides whether it is a satisfying assignment, and we find the assignment itself more interesting than the fact that it is or is not satisfying. In game theory we find the move chosen by a player more interesting than the outcome that results from that move. The author and collaborators are developing a theory of games in which selection functions are viewed as generalised notions of rationality, used to model players. By realising that strategic contexts in game theory are examples of continuations we can see that classical game theory narrowly misses being in CPS, and that a small change of viewpoint yields a theory of games that is better behaved, and especially more compositional.

Contents

1 Introduction

A selection function is a type-2 function $\varepsilon : \mathscr{J}X$ where

$$\mathscr{J}X = (X \to R) \to X$$

We can view the input $k : X \to R$ to such a function in several ways: as a generalised predicate, as the context of a decision, or as a continuation. These are emphasised respectively in topology [4], game theory [5] and proof theory [7]. The earliest selection functions considered were computable instances of Hilbert's ε-operator $(X \to \mathbb{B}) \to X$, which witness a computational form of topological compactness. An operation $\otimes : \mathscr{J}X \times \mathscr{J}Y \to \mathscr{J}(X \times Y)$, which is the (left-leaning) monoidal product of the strong monad $\mathscr{J}$, witnesses the fact that the product of two compact spaces is compact. Remarkably this extends to countable products, leading to the derivation of 'seemingly impossible functional programs' [3] that search set-theoretically infinite but topologically compact types such as $\mathbb{N} \to \mathbb{B}$ (the Cantor space) in a finite amount of time.

It was noticed by Paulo Oliva that the product of selection functions is equivalent to Spector's bar recursion [8], a notoriously obscure computational feature used to realise the axiom of countable choice via a double negative translation and Gödel's Dialectica interpretation. Bar recursion is important in the proof mining programme because it can be used to interpret proofs of classical analysis (including differential equations and ergodic theory) [14], but the computer programs arising from such proofs via bar recursion, while provably correct, are not well suited to human understanding.

The next step made by Escardó and Oliva was the connection with game theory, by realising that economic rationality is modelled by the selection function

$$\arg\max : (X \to \mathbb{R}) \to X$$

which finds a point maximising a real-valued function (say, on a finite set X). By generalising selection functions by replacing the booleans with $\mathbb{R}$ or an arbitrary type R, the product of selection functions is seen to correspond to an well-known and intuitive algorithm in game theory known as backward induction [5]. Since proof interpretations using bar recursion can be rewritten using the product of selection functions, we therefore obtain a computational interpretation of proofs in classical analysis that is amenable to human understanding via game theory [16].

The relationship to the continuation monad is immediately obvious: the Hilbert ε-operator is a refined form of the quantifier

$$\exists : (X \to \mathbb{B}) \to \mathbb{B}$$

and the operator $\arg\max$ is a refined form of

$$\max : (X \to \mathbb{R}) \to \mathbb{R}$$

These functions are both computations in the continuation monad

$$\mathscr{K}X = (X \to R) \to R$$

Escardó and Oliva call any function with a type of this form a *quantifier*. Moreover the relationships

$$\exists p = p(\varepsilon p)$$

and

$$\max p = p(\arg\max p)$$

define a monad morphism from the selection to the continuation monad. Thus in proof theory the J-translation (or Peirce translation) [7] can be seen as a refined form of the usual (generalised) double negative translation, and it would be tempting to view the selection monad as a refinement of the continuation monad. However things are not so clear, because call/cc of the selection monad has a more specific type, and behaves differently.

In section 2 we will explain the (quite complex) bind operation of the selection monad using intuition from CPS, which previously has only been derived in a purely formal way. In sections 3 and 4 we will discuss the call/cc-like operator that exists for the selection monad, and see its behaviour experimentally. Finally in section 5 we will informally discuss ongoing work by the author and others on applications of selection functions and CPS in game theory.

2 Two monads for CPS

We begin by recalling the usual intuition for the continuation monad, which can be used to implement (delimited) continuations in a pure language such as Haskell. We view all computation as being done relative to a *continuation*, which takes the return value and chains it to the future of the computation. Thus our computation has the shape[1]

$$\cdots \longrightarrow X \xrightarrow{\quad k \quad} R$$

We will call X the type of *intermediate values*, and R the type of *final values*. The key idea of the continuation monad is that we always work relative to an *unknown* continuation k, although we may fix the type R. (The ability to fix R may have begun as a quirk due to Hindley-Milner typing, see [13], but it is vital to many applications of selection functions.) We allow our functions to have side-effects, modelled by functions $X \to MR$ for a suitable monad M. A computation in the continuation monad is a function with type

$$\mathscr{K}_R^M = (X \to MR) \to MR$$

which computes a final result given a continuation.

[1] The diagrams in this paper are not intended to be formal, but rather as a possible aid to intuition, which not every reader will find helpful. 'Plain' arrows are intended to live in the Kleisli category of the base monad M, while arrows with quote marks around the name live in the Kleisli category of either $\mathscr{K}_R^M$ or $\mathscr{J}_R^M$.

To embed a pure value $x : X$ we simply view it as the intermediate value, and apply the continuation to it immediately:

$$\eta_{\mathscr{K}_R^M} x = \lambda k^{X \to MR}.kx$$

For the bind, suppose we have a computation $\varphi : \mathscr{K}_R^M X$ and a family of computations $F : X \to \mathscr{K}_R^M Y$. Equivalently, F is a computation $X \to Y$, which we can run by supplying it with a continuation from Y. We form a computation that has the shape

$$\cdots \longrightarrow X \xrightarrow{\text{“}F\text{”}} Y \xrightarrow{\quad k \quad} R$$

The key to understanding the bind of both the continuation and selection monads is that we have two views of a computation like this: we can either view the intermediate result as being at X or Y. The 'external' view of $\varphi \gg\!=_{\mathscr{K}_R^M} F : \mathscr{K}_R^M Y$ is that it is a computation with the intermediate result being at Y. Suppose we are running this computation, so we have a continuation $k : Y \to MR$. To find the final result we move to the other view, and run the computation φ by building it a longer continuation $k' : X \to MR$. Given an intermediate value $x : X$ we can now find the final result $k'x$ because we have the continuation k from Y: it is $k'x = Fxk$. Therefore the bind operator for the continuation monad is given by

$$\varphi \gg\!=_{\mathscr{K}_R^M} F = \lambda k^{Y \to MR}.\varphi \lambda x^X.Fxk$$

This intuition for the continuation monad transfers directly to the selection monad. Whereas a computation in the continuation monad computes the final result given a continuation, a computation in the selection monad computes the intermediate result instead. Thus the selection monad is

$$\mathscr{J}_R^M X = (X \to MR) \to MX$$

First, suppose we want to embed a pure value $x : X$ as the intermediate result. Given a continuation $k : X \to MR$, we ignore the continuation and simply return x, embedded as a computation in M:

$$\eta_{\mathscr{J}_R^M} x = \lambda k^{X \to MR}.\eta_M x$$

For the bind operator of the selection monad we must again move between the two views of the computation

$$\cdots \longrightarrow X \xrightarrow{\text{“}F\text{”}} Y \xrightarrow{\quad k \quad} R$$

Suppose we have a computation $\varepsilon : \mathscr{J}_R^M X$ and a family of computations $F : X \to \mathscr{J}_R^M Y$. To run $\varepsilon \gg\!=_{\mathscr{J}_R^M} F : \mathscr{J}_R^M Y$ we are given a continuation $k : Y \to MR$, and we must return an intermediate result at Y. Our first task is to find a way to turn an intermediate result at X into one at Y. Suppose

we have an intermediate value $x : X$, so we have a computation $Fx : \mathscr{J}_R^M Y$ with intermediate value at Y. We can now run this computation with our continuation k, which yields an intermediate value at Y. Thus we have a function $f : X \to MY$ given by

$$fx = Fxk$$

Now we can extend our continuation k with f to give a continuation k' from X, namely

$$k'x = fx \gg\!=_M k = Fxk \gg\!=_M k$$

By running the computation ε with this continuation, we obtain an intermediate result at X. Finally, we must apply the function f again to obtain an intermediate result at Y, which is what we need. Written out explicitly, the operator is

$$\varepsilon \gg\!=_{\mathscr{J}_R^M} F = \lambda k^{Y\to MR}.(\varepsilon\lambda x^X.Fxk \gg\!=_M k) \gg\!=_M \lambda x^X.Fxk$$

Obviously, the sheer complexity of this formula makes reasoning about programs in the selection monad very difficult. In practice, Haskell is a very useful tool for qualitatively understanding the behaviour of such programs.

No published proof of the monad laws for the selection monad exists. The original proof (in the absence of other side-effects) was generated by an equational reasoning tool written by Martin Escardó specifically for this purpose, resulting in several pages of formal manipulations. For an arbitrary side-effects the unit laws were checked by the author with several pages of equational reasoning, including use of functional extensionality (η-expansion). The associative law seems to be impractical to check by hand, but the proof is found by Coq's tactic for intuitionistic logic. However previously the unit and bind operators were considered purely formal objects (derived simply by proving them as theorems of intuitionistic logic), and the intuitions developed in this section suggest for the first time that it might be possible to find a human-readable proof, by reducing to the monad laws for the continuation monad.

3 Call/cc for the selection monad

For building computations in the continuation monad we can use the call-with-current-continuation operator. This has type

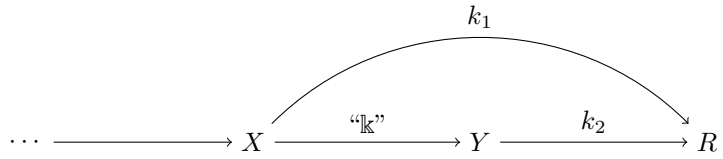
$$\text{æ} : ((X \to \mathscr{K}_R^M Y) \to \mathscr{K}_R^M X) \to \mathscr{K}_R^M X$$

The input Φ to æ is called a *continuation handler*, which is a computation that has access to the *current continuation* $\Bbbk : X \to \mathscr{K}_R^M Y$ (we will use the letter k to refer to an actual continuation, and $\Bbbk$ to refer to a continuation reified as a computation in the continuation or selection monad). The current continuation is just a computation $X \to Y$ in the continuation monad, and the purpose of æΦ is to call $\Phi\Bbbk$ where $\Bbbk$ is bound to the continuation of æΦ. The implementation of æ is given by

$$\text{æ}\Phi = \lambda k_1^{X\to MR}.\Phi(\lambda x^X, k_2^{Y\to MR}.k_1 x)k_1$$

Given our description it is quite easy to read this formula. The parameter given to Φ should be the continuation k_1 reified as a function $X \to K_R^M Y$. We have exactly that: given an input, we make the computation which ignores its own continuation and uses k_1 instead.

Put another way, we want to build a computation $\Bbbk$ to which we can apply Φ. Consider the diagram



In this diagram, X is the intermediate type at the point in the handler at which the current continuation is invoked, and Y is the intermediate result when the handler ends. We do not have a pure function $X \to Y$, but we can build a CPS computation $\Bbbk : X \to \mathscr{K}_R^M Y$ by short-cutting with the continuation k_1, that is,

$$\Bbbk x = \lambda k_2^{Y \to MR}.k_1 x$$

However, the principal type of the term $\mathfrak{c}$ is the far more general

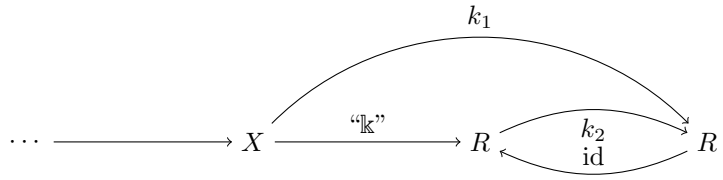
$$((A \to B \to C) \to (A \to C) \to D) \to (A \to C) \to D$$

We obtain the specific type $((X \to \mathscr{K}_R^M Y) \to \mathscr{K}_R^M X) \to \mathscr{K}_R^M X$ by setting $A = X$, $B = Y \to MR$ and $C = D = MR$. In order that our computations are in the selection rather than the continuation monad, we instead set $C = MR$ and $D = MX$. This implies that $Y = R$, and so $B = R \to MR$. Thus $\mathfrak{c}$ has the type

$$((X \to \mathscr{J}_R^M R) \to \mathscr{J}_R^M X) \to \mathscr{J}_R^M X$$

This is the call-with-current-continuation operator for the selection monad. It is equal (as an untyped λ-term) to the ordinary call/cc, and the difference in behaviour comes from the different bind operator with which it is composed.

If we draw a similar diagram we get



We have the continuation k_1 which gives us a final result given an intermediate result at X, which is also the point at which the continuation is invoked in the handler. The computation $\Bbbk$, given its continuation k_2, should return the intermediate result. However instead we simply return $k_1 x$, which is really the final result, which we arranged to have the same type. That is, we are implicitly

assuming that k_2 is always the trivial continuation, even though it might not be. We could change the type to

$$((X \to \mathscr{K}_R^M R) \to \mathscr{J}_R^M X) \to \mathscr{J}_R^M X$$

to emphasise that $\Bbbk$ returns a final rather than an intermediate result, but that only moves our dishonesty elsewhere because to actually invoke the current continuation in the handler we would need an 'invoke-continuation' function with type $\mathscr{K}_R^M R \to \mathscr{J}_R^M R$, which treats its final result as intermediate.

In the next section we will see experimental results about the operational behaviour of this function, including when the whole computation is run with a nontrivial continuation.

4 Programming with selection call/cc

This section uses the author's implementation of the selection monad transformer from [10], and assumes some familiarity with monad transformers and the syntax of Haskell. As an example we will take the simple CPS Haskell program

```
trace :: (MonadIO m) => String -> m ()
trace = liftIO . putStrLn

foo :: ContT r IO Int
foo = do trace "In foo"
         n <- callCC $ \k -> do trace "In handler"
                                m <- k 0
                                trace "Still in handler"
                                return (m + 1)
         trace "In continuation"
         return (n + 1)
```

When run interactively we obtain

```
ghci> runContT foo return
In foo
In handler
In continuation
1
```

This program demonstrates an important fact about call/cc: once the continuation is run, control is never returned to the handler. Thus while the continuation logically returns a value and we can bind it to a variable, any code after the continuation is called is unreachable. This is in contrast to the selection monad, which we will see next.

The same program written in the selection monad is

```
bar :: SelT Int IO Int
bar = do trace "In bar"
         n <- callCC' $ \k -> do trace "In handler"
                                 m <- k 0
                                 trace "Still in handler"
                                 return (m + 1)
         trace "In continuation"
         return (n + 1)
```

By running this program we can see the execution trace:

```
ghci> runSelT bar return
In bar
In handler
In continuation
Still in handler
In continuation
3
```

As can be seen, there is a coroutine-like dialogue between the handler and the continuation, with data flowing back and forth.

Like in the continuation monad, the continuation can be called from arbitrarily far inside a call stack. In particular we can write a product of selection functions that has access to its current continuation. For example, we can write a SAT solver that calls the current continuation with a dummy input once per iteration:

```
sat :: Int -> SelT Bool IO [Bool]
sat n = do bs <- callCC' $ \k -> sequence $ replicate n $
             do b <- SelT ($ True)
                liftIO $ putStr $ "b = " ++ show b ++ ", "
                k []
                return b
           trace $ "Continuation called with " ++ show bs
           return bs
```

This program is based on the verbose SAT solver in [10], and is designed to exhibit as many unexplained patterns as possible. Given a particular formula like

```
f :: [Bool] -> IO Bool
f bs = return $ bs!!0 && not(bs!!1) && bs!!2
```

we can run it by

```
ghci> runSelT (sat 3) f
b = True, Continuation called with []
b = True, Continuation called with []
b = True, Continuation called with []
```

```
Continuation called with [True,True,True]
b = False, Continuation called with []
Continuation called with [True,True,False]
b = False, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
b = True, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,True,True]
b = False, Continuation called with []
Continuation called with [True,True,False]
b = False, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
Continuation called with [True,False,True]
[True,False,True]
```

(Note that each time the current continuation is called with the empty list it also calls the formula f, but the expressions which index the empty list, which would result in a runtime exception, are not evaluated due to the lazy semantics of Haskell.) It is already an open question how to explain patterns such as $TTT, TTF, TFT, TFT, TTT, TTF, TFT, TFT$ resulting from the operational behaviour of the product of selection functions. The positions of the empty list in this trace only seem to deepen the mystery.

5 Strategic contexts are continuations

The author and collaborators are developing a theory of games based on selection functions, which will be informally described in this section. In a computable game theory, the computations done by players are naturally CPS computations, in the sense that the computation divides into two parts: first the player computes a move, and then the rules of the game (and the other players) use the move to compute the outcome. However the player's computation of a move is done with knowledge of how that move will be used to compute an outcome, which makes it a CPS computation. In classical game theory a player may have a finite set of choices X, and each choice determines a real number $kx : \mathbb{R}$ called a *utility*. A fundamental assumption of classical game theory is that rational players act so as to maximise their utility. We can view the continuation of the player's decision as $k : X \to \mathbb{R}$, and the computation done by the player is $\arg\max : \mathscr{J}_{\mathbb{R}}X$. One reason that this is interesting is that the product of selection functions computes Nash equilibria of sequential games of

perfect information, by implementing an algorithm known as backward induction [6]. Thus what was apparently a fact of applied mathematics, concerning the behaviour of interacting rational agents, in fact arises very naturally in a theoretical setting.

An immediate application of these ideas is that we can easily generalise some concepts in classical game theory to arbitrary selection functions, which allows us to work with players who are not classically rational without having to build a new game theory from scratch. This is applied to 'context-dependent choice' in voting games in [11], where it is shown that the fixpoint operator $\mathrm{fix} : \mathscr{J}_X^{\mathcal{P}} X$ models a 'Keynes agent' who would like to vote for the winner of an election, or a so-called Keynes beauty contest. Similarly the operator that selects non-fixpoints models a 'punk' who would like to vote for anybody but the winner, which corresponds to a one-shot minority game [12]. In general, a 'context-dependent agent' may have preferences not just over outcomes, but over the ways in which those outcomes are achieved. This formally gives no new expressive power because the type of outcomes could be extended to include all relevant information, but selection games are more convenient, and in particular seem to be more scalable. To give a silly example, a million dollars earned is not equivalent to a million dollars stolen. To model such 'social concerns' classically, an explicit conversion rate between morality and dollars must be given globally and encoded into the outcome structure of the game. With selection functions this can be done instead on a per-agent basis.

We conclude by giving several research directions currently being explored by the author and several collaborators. In each of these, the slogan 'strategic contexts are continuations' is an important part of the author's intuitive understanding.

- Most importantly, the author has recently developed a graphical calculus for game theory by extending string diagrams for monoidal categories [17]. The semantics of these string diagrams uses CPS very heavily: a diagram generally represents some group of interacting agents, which is equivalent to a 'generalised agent' with preferences over both strategies and continuations from computations of strategies. The categorical composition and tensor product (which are primitive forms of sequential and parallel composition, respectively) are both defined using (delimited) continuation transformers, and cannot be written in a direct style.

- The way in which this graphical calculus differs from the ones used in quantum theory is the way in which backward-causality is treated. Game theory contains a quite restricted form of backward-causality due to agents reasoning about future events. An agent is graphically connected to a relevant future value by a feedback-like operation. Intriguingly there is an extremely close analogy with shift/reset operators [1] here: a decision made by an agent is like 'shift' (it captures a continuation), and the point in the future at which a value is designated as the outcome for an agent is like 'reset' (it delimits a continuation). In the graphical calculus

these also appear strongly analogous to the unit and counit of a compact structure in category theory, which correspond to pair-production and pair-annihilation in quantum mechanics.

- The first two points relate to defining games and solution concepts, but not to computing solutions in general. In this direction, the author has used various monad transformer stacks with the selection monad at the top. For sequential games of perfect information this can be done in generality using the product of selection functions, but to extend to simultaneous and other games must be done on a per-effect basis. The most progress has been made for nondeterministic games, in which agents can make nondeterministic choices between several moves. (True nondeterminism is essentially unknown in economic game theory; for a use in game theory in computer scientist see [15, chapter 9].) The idea is to define a new 'sum of selection functions' operator

$$\oplus : \mathscr{J}_R^M X \times \mathscr{J}_R^M Y \to \mathscr{J}_R^M (X \times Y)$$

  which is analogous to the product of selection functions, but for simultaneous games. Games with nondeterministic strategies are noticeably better behaved than either pure or mixed strategies, with solution spaces carrying more structure.

- For infinite games, and games with mixed strategies, things are more interesting and difficult. The author is bringing together many ideas from functional programming, topology and category theory to attack these problems. One starting point is that probability distributions form a monad [9, 2] which carries additional topological structure.

Acknowledgments: The author is grateful to the anonymous reviewers for their comments, and also gratefully acknowledges EPSRC grant EP/K50290X/1 which funded this work.

References

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. *Proceedings of the 1990 ACM conference on Lisp and functional programming*, pages 151–160, 1990. doi: 10.1145/91556.91622. 9

[2] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in Haskell. *Journal of functional programming*, 16(1):21–34, 2006. doi: 10.1017/S0956796805005721. 10

[3] Martin Escardó. Seemingly impossible functional programs, 2007. URL `http://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/`. 1

[4] Martin Escardó. Exhaustible sets in higher-type computation. *Logical methods in computer science*, 4(3:3):1–37, 2008. doi: 10.2168/lmcs-4(3:3)2008. 1

[5] Martin Escardó and Paulo Oliva. Sequential games and optimal strategies. *Proc R Soc A*, 467:1519–1545, 2011. doi: 10.1098/rspa.2010.0471. 1

[6] Martin Escardó and Paulo Oliva. Computing Nash equilibria of unbounded games. *Proceedings of the Turing centenary conference*, 2012. 9

[7] Martin Escardó and Paulo Oliva. The Pierce translation. *Annals of pure and applied logic*, 163(6):681–692, 2012. 1, 2

[8] Martin Escardó and Paulo Oliva. Bar recursion and products of selection functions. To appear, 2014. 1

[9] Michèlle Giry. A categorical approach to probability theory. *Categorical aspects of topology and analysis*, pages 68–85, 1982. doi: 10.1007/BFb0092872. 10

[10] Jules Hedges. Monad transformers for backtracking search. In Paul Levy and Neel Krishnaswami, editors, *Proceedings of the 5th workshop on mathematically structured functional programming*, volume 153 of *Electronic proceedings in theoretical computer science*, pages 31–50. Open Publishing Association, 2014. doi: 10.4204/EPTCS.153.3. 6, 7

[11] Jules Hedges, Paulo Oliva, Evguenia Winschel, Viktor Winschel, and Philipp Zahn. A higher-order framework for decision problems and games. Submitted, 2014. 9

[12] Willemien Kets. Learning with fixed rules: the minority game. *Journal of economic surveys*, 26(5):865–878, 2011. doi: 10.1111/j.1467-6419.2011.00686.x. 9

[13] Oleg Kiselyov. Undelimited continuations are co-values rather than functions. URL `http://okmij.org/ftp/continuations/undelimited.html`. 2

[14] Ulrich Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer, 2008. 1

[15] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006. 10

[16] Paulo Oliva and Thomas Powell. A game-theoretic computational interpretation of proofs in classical analysis. To appear, 2012. 1

[17] Peter Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New structures for physics*, pages 289–355. Springer, 2011. 9

A Modular Structural Operational Semantics for Delimited Continuations

Neil Sculthorpe Paolo Torrini Peter D. Mosses

PLanCompS Project
Department of Computer Science
Swansea University
Swansea, UK

Abstract

It has been an open question as to whether the Modular Structural Operational Semantics framework can express the dynamic semantics of *call/cc*. This paper shows that it can, and furthermore, demonstrates that it can express the more general delimited control operators *control* and *shift*.

Contents

List of Figures

1 Introduction

Modular Structural Operational Semantics (MSOS) [23, 24, 25] is a variant of the well-known Structural Operational Semantics (SOS) framework [27]. The principal innovation of MSOS relative to SOS is that it allows the semantics of a programming construct to be specified independently of any auxiliary entities with which it does not directly interact. For example, function application can be specified by MSOS rules without mentioning stores or exception propagation.

While it is known that MSOS can specify the semantics of programming constructs for exception handling [7, 8, 23], it has been unclear whether MSOS can specify more complex control-flow operators, such as *call/cc* [1, 9]. Indeed, the perceived difficulty of handling control operators has been regarded as one of the main limitations of MSOS relative to other modular semantic frameworks (e.g. [28, Section 2]). This paper demonstrates that the dynamic semantics of *call/cc can* be specified in MSOS, with no extensions to the MSOS framework required. We approach this by first specifying the more general delimited control operators *control* [16, 17, 32] and *shift* [11, 12, 13], and then specifying *call/cc* in terms of *control*. In contrast to most other operational specifications of control operators given in direct style (e.g. [16, 20, 22, 31]), ours are based on labelled transitions, rather than on evaluation contexts.

We will begin by giving a brief overview of delimited continuations (Section 2) and MSOS (Section 3). The material in these two sections is not novel, and can be skipped by a familiar reader.

2 Delimited Continuations

At any point in the execution of a program, the *current continuation* represents the rest of the computation. In a meta-language sense, a continuation can be understood as a context in which a program term can be evaluated. *Control operators* allow the current continuation to be treated as an object in the language, by reifying it as a first-class abstraction that can be applied and manipulated. The classic example of a control operator is *call/cc* [1, 9].

Delimited continuations generalise the notion of a continuation to allow representations of partial contexts, relying on a distinction between inner and outer context. Control operators that manipulate delimited continuations are always associated with *control delimiters*. The most well-known delimited control operators are *control* (associated with the *prompt* delimiter) [16, 17, 32] and *shift* (associated with the *reset* delimiter) [11, 12, 13], both of which can be used to simulate *call/cc*. The general idea of *control* and *shift* is to capture the current continuation up to the innermost enclosing delimiter, representing the inner context. We will give an informal description of *control* in this section. The formal MSOS specification of *control* is given in Section 4, where we also specify *shift* and *call/cc* in terms of *control*.

control is a (call-by-value) unary operator that takes a higher-order function f as its argument, where f expects a reified continuation as its argument. When

executed, *control* reifies the current continuation, up to the innermost enclosing *prompt*, as a function k. That inner context is then discarded, and replaced with the application $f\ k$. Other than its interaction with *control*, *prompt* is simply a unary operator that evaluates its argument and returns the resulting value.

Let us consider some examples. In the following expression, the continuation k is bound to the function $(\lambda x.\ 2 * x)$, the result of the *prompt* application is 14, and thus the final result is 15:

$$1 + \mathit{prompt}(2 * \mathit{control}(\lambda k.\ k\ 7)) \quad \rightsquigarrow \quad 15$$

A reified continuation can be applied multiple times, for example:

$$1 + \mathit{prompt}(2 * \mathit{control}(\lambda k.\ k(k\ 7))) \quad \rightsquigarrow \quad 29$$

Furthermore, a continuation need not be applied at all. For example, in the following expression, the multiplication by two is discarded:

$$1 + \mathit{prompt}(2 * \mathit{control}(\lambda k.\ 7)) \quad \rightsquigarrow \quad 8$$

In the preceding examples, the continuation k could have been computed statically. However, in general, the current continuation is the context at the point in a program's execution when *control* is executed, by which time some of the computation in the source program may already have been performed. For example, the following program will print ABB:

$$\mathit{prompt}(\ \mathit{print}\ 'A'\ ;\ \mathit{control}(\lambda k.\ (k\ ()\ ;\ k\ ()))\ ;\ \mathit{print}\ 'B'\ ) \quad \rightsquigarrow \quad ABB$$

The command $(\mathit{print}\ 'A')$ is executed before the *control* operator, so does not form part of the continuation reified by *control*. In this case, k is bound to $(\lambda x.\ (x\ ;\ \mathit{print}\ 'B'))$, and so B is printed once for every application of k.

Further examples of *control* can be found in the online test suite accompanying this paper [30], and in the literature [16, 17].

3 Modular SOS

The rules in this paper will be presented using *Implicitly Modular SOS* (I-MSOS) [25], a variant of MSOS that has a notational style similar to conventional SOS. I-MSOS can be viewed as syntactic sugar for MSOS. We assume the reader is familiar with SOS (e.g. [3, 27]) and the basics of MSOS [23, 24, 25].

The key notational convenience of I-MSOS is that any auxiliary entities (e.g. stores or environments) that are not mentioned in a rule are *implicitly propagated* between the premise(s) and conclusion, allowing entities that do not interact with the programming construct being specified to be omitted from the rule. Two types of entities are relevant to this paper: inherited entities (e.g. environments), which, if unmentioned, are implicitly propagated from the conclusion to the premises, and observable entities (emitted signals, e.g. exceptions), which, if unmentioned, are implicitly propagated from a sole premise to

$$\frac{E \to E'}{\mathbf{throw}(E) \to \mathbf{throw}(E')} \tag{1}$$

$$\frac{val(V)}{\mathbf{throw}(V) \xrightarrow{\mathsf{exc}\ \mathbf{some}(V)} \mathbf{stuck}} \tag{2}$$

$$\frac{E \xrightarrow{\mathsf{exc}\ \mathbf{none}} E'}{\mathbf{catch}(E, H) \xrightarrow{\mathsf{exc}\ \mathbf{none}} \mathbf{catch}(E', H)} \tag{3}$$

$$\frac{E \xrightarrow{\mathsf{exc}\ \mathbf{some}(V)} E'}{\mathbf{catch}(E, H) \xrightarrow{\mathsf{exc}\ \mathbf{none}} \mathbf{apply}(H, V)} \tag{4}$$

$$\frac{val(V)}{\mathbf{catch}(V, H) \to V} \tag{5}$$

Figure 1: I-MSOS rules for exception handling.

the conclusion. Observable entities are required to have a default value, which is implicitly used in the conclusion of rules that lack a transition-rule premise and do not mention the entity.

To demonstrate the specification of control operators using I-MSOS rules, this paper will use the *funcon framework* [8]. This framework contains an open collection of modular *fundamental constructs* (funcons), each of which has its semantics specified independently by I-MSOS rules. The framework is designed to serve as a target language for semantic specifications of programming languages, intended to be specified by an inductive translation in the style of denotational semantics. However, this paper is not concerned with the translation of control operators from any specific language: our aim is to give MSOS specifications of control operators, and the funcon framework is a convenient environment for specifying prototypical control operators. Examples of translations into funcons can be found in [8, 26].

We will now present some examples of funcons, and their specifications as small-step I-MSOS rules. We typeset funcon names in **bold**, meta-variables in capitalised *italic*, and the names of auxiliary entities in sans-serif. No familiarity with the funcon framework is required: for the purposes of understanding this paper the funcons may simply be regarded as abstract syntax.

Figure 1 presents I-MSOS rules for the exception-handling funcons **throw** and **catch** [8]. The idea is that **throw** emits an exception signal, and **catch** detects and handles that signal. The first argument of **catch** is the expression to be evaluated, and the second argument (a function) is the exception handler. Exception signals use an observable entity named exc, which is written as a label on the transition arrow. The exc entity has either the value **none**, denoting the

$$val(\mathbf{closure}(\rho, I, E)) \tag{6}$$

$$\frac{\rho(I) = V}{\mathsf{env}\ \rho \vdash \mathbf{bv}(I) \to V} \tag{7}$$

$$\mathsf{env}\ \rho \vdash \mathbf{lambda}(I, E) \to \mathbf{closure}(\rho, I, E) \tag{8}$$

$$\frac{E_1 \to E_1'}{\mathbf{apply}(E_1, E_2) \to \mathbf{apply}(E_1', E_2)} \tag{9}$$

$$\frac{val(V) \qquad E \to E'}{\mathbf{apply}(V, E) \to \mathbf{apply}(V, E')} \tag{10}$$

$$\frac{val(V) \qquad \mathsf{env}\ (\{I \mapsto V\}/\rho) \vdash E \to E'}{\mathsf{env}\ \_ \vdash \mathbf{apply}(\mathbf{closure}(\rho, I, E), V) \to \mathbf{apply}(\mathbf{closure}(\rho, I, E'), V)} \tag{11}$$

$$\frac{val(V_1) \qquad val(V_2)}{\mathbf{apply}(\mathbf{closure}(\rho, I, V_1), V_2) \to V_1} \tag{12}$$

Figure 2: I-MSOS rules for lambda calculus.

absence of an exception, or $\mathbf{some}(V)$, denoting the occurrence of an exception with value V. The premise $val(V)$ requires the term V to be a value, thereby controlling the order in which the rules can be applied. In the case of **throw**, first the argument is evaluated to a value (Rule 1), and then an exception carrying that value is emitted (Rule 2). In the case of **catch**, the first argument E is evaluated while no exception occurs (Rule 3). If an exception does occur, then the handler H is applied to the exception value and the computation E is abandoned (Rule 4). If E evaluates to a value V, then H is discarded and V is returned (Rule 5).

Observe that rules 1 and 5 do not mention the exc entity. In Rule 1 it is implicitly propagated from premise to conclusion, and in Rule 5 it implicitly has the default value **none**. Also observe that none of the rules in Figure 1 mention any other entities such as environments or stores; any such entities are also implicitly propagated.

Figure 2 presents I-MSOS rules for identifier lookup (**bv**, "bound-value"), abstraction (**lambda**), and application (**apply**). Note that the **closure** funcon is a *value constructor* [7] (specified by Rule 6), and thus has no transition rules of its own. We present these rules here for completeness, as these funcons will be used when defining the semantics of control operators in Section 4.

4 I-MSOS Specifications of Control Operators

We now present a dynamic semantics for control operators in the MSOS framework. We will specify *control* and *prompt* directly, and then specify *shift*, *reset* and *call/cc* in terms of *control* and *prompt*. Our approach is signal-based in a similar manner to the I-MSOS specifications of exceptions (Figure 1): a control operator emits a signal when executed, and a delimiter catches that signal and handles it. Note that there is no implicit top-level delimiter around a funcon program—a translation to funcons from a language that does have an implicit top-level delimiter should insert an *explicit* top-level delimiter.

4.1 Overview of our Approach

We represent reified continuations as first-class abstractions, using the **lambda** funcon from Section 3. However, we do not maintain an explicit representation of the *current* continuation in our semantics; instead, our approach is to construct the continuation from the program term whenever a control operator is executed. We achieve this by exploiting the way that a small-step semantics, for each step of computation, traverses the program term from the root to the current operation. Thus, for any step at which a control operator is executed, not only will a rule for the control operator be applied, but so too will a rule for the enclosing delimiter. At each such step, the current continuation of the control operator corresponds to an abstraction of that operator from the sub-term of the enclosing delimiter, and thus can be constructed from that sub-term. This is achieved in two stages: the rule for the control operator replaces the occurrence of the control operator with a fresh identifier, and the rule for the delimiter constructs the abstraction from the updated sub-term.

At a first approximation, this suggests the following rules:

$$\frac{\textit{fresh-id}(I)}{\textbf{control}(F) \xrightarrow{\text{control } \textbf{some}(F,I)} \textbf{bv}(I)} \tag{13}$$

$$\frac{E \xrightarrow{\text{control } \textbf{some}(F,I)} E' \qquad K = \textbf{lambda}(I, E')}{\textbf{prompt}(E) \xrightarrow{\text{control } \textbf{none}} \textbf{prompt}(\textbf{apply}(F, K))} \tag{14}$$

The premise *fresh-id*(I) requires that I be a fresh identifier. Rule 13 replaces the term **control**(F) with **bv**(I), and emits a signal containing the function F and the identifier I. The signal is then caught and handled by **prompt** in Rule 14. The abstraction K representing the continuation of the executed control operator is constructed by combining I with the updated sub-term E' (which will now contain **bv**(I) in place of **control**(F)). Note that although the signal entity is named control, this name brings no inherent connection to the funcon **control**, as entities live in a separate namespace to funcons.

$$\frac{\rho(I) = V}{\textsf{meta-env}\ \rho \vdash \textbf{meta-bv}(I) \to V} \tag{15}$$

$$\frac{E_1 \to E_1'}{\textbf{meta-let-in}(I, E_1, E_2) \to \textbf{meta-let-in}(I, E_1', E_2)} \tag{16}$$

$$\frac{\mathit{val}(V) \qquad \textsf{meta-env}\ (\{I \mapsto V\}/\rho) \vdash E \to E'}{\textsf{meta-env}\ \rho \vdash \textbf{meta-let-in}(I, V, E) \to \textbf{meta-let-in}(I, V, E')} \tag{17}$$

$$\frac{\mathit{val}(V_1) \qquad \mathit{val}(V_2)}{\textbf{meta-let-in}(I, V_1, V_2) \to V_2} \tag{18}$$

Figure 3: I-MSOS rules for meta-environment bindings.

4.2 The Meta-environment

There is one problem with the approach we have just outlined, which is that the fresh identifier I is introduced dynamically when the control operator executes, by which time closures may have already been formed. In particular, if **control** occurs inside the body of an applied closure, and the enclosing **prompt** is outside that closure, then the $\textbf{bv}(I)$ funcon that is introduced by Rule 13 would be evaluated in a closed environment that does not contain a binding for I.

To address this, we will make use of an auxiliary environment called meta-env (meta-environment). This environment is used for bindings that should not interact with bindings in the standard environment, such as via shadowing or being captured in closures. In this paper, we will use the meta-environment to essentially achieve the same effect as substitution (MSOS does not provide a substitution operation, relying instead on environments). Figure 3 specifies $\textbf{meta-bv}(I)$, which looks up the identifier I in the meta-environment, and $\textbf{meta-let-in}(I, V, E)$, which binds the identifier I to the value V in the meta-environment, and scopes that binding over the expression E. We will make use of these funcons in the next subsection, where we give our complete specification of **control** and **prompt**.

4.3 Dynamic Semantics of *control* and *prompt*

We specify **control** as follows:

$$\frac{E \to E'}{\textbf{control}(E) \to \textbf{control}(E')} \tag{19}$$

$$\frac{\mathit{val}(F) \qquad \mathit{fresh\text{-}id}(I)}{\textbf{control}(F) \xrightarrow{\textsf{control}\ \textbf{some}(F,I)} \textbf{meta-bv}(I)} \tag{20}$$

Rule 19, in combination with the $\mathit{val}(F)$ premise on Rule 20, ensures that the argument function is evaluated to a closure before Rule 20 can be applied. Notice that Rule 20, in contrast to the preliminary Rule 13, uses **meta-bv** to lookup I in the meta-environment.

We then specify **prompt** as follows:

$$\frac{\mathit{val}(V)}{\mathbf{prompt}(V) \to V} \tag{21}$$

$$\frac{E \xrightarrow{\text{control } \mathbf{none}} E'}{\mathbf{prompt}(E) \xrightarrow{\text{control } \mathbf{none}} \mathbf{prompt}(E')} \tag{22}$$

$$\frac{E \xrightarrow{\text{control } \mathbf{some}(F,I)} E' \qquad K = \mathbf{lambda}(I, \mathbf{meta\text{-}let\text{-}in}(I, \mathbf{bv}(I), E'))}{\mathbf{prompt}(E) \xrightarrow{\text{control } \mathbf{none}} \mathbf{prompt}(\mathbf{apply}(F, K))} \tag{23}$$

Rule 21 is the case when the argument is a value; the **prompt** is then discarded. Rule 22 evaluates the argument expression while no control signal is being emitted by that evaluation. Rule 23 handles the case when a control signal is detected, reifying the current continuation and passing it as an argument to the function F. Notice that, in contrast to the preliminary Rule 14, I is rebound in the meta-environment.

Rules 19–23 are our complete I-MSOS specification of the dynamic semantics of **control** and **prompt**, relying only on the existence of the lambda-calculus and meta-environment funcons from figures 2 and 3. These rules are modular: they are valid independently of whether the control operators coexist with a mutable store, exceptions, or other effectful programming constructs. Our rules correspond closely to those in specifications of *control* and *prompt* based on evaluation contexts [16, 22]. However, our specifications communicate between *control* and *prompt* by emitting signals, and thus do not require evaluation contexts.

4.4 Dynamic Semantics of *shift* and *reset*

The *shift* operator differs from *control* in that every application of a reified continuation is implicitly wrapped in a delimiter, which has the effect of separating the context of that application from its inner context [5]. This difference between *control* and *shift* is comparable to that between dynamic and static scoping, insofar as with *shift*, the application of a reified continuation cannot access its context, in the same way that a statically scoped function cannot access the environment in which it is applied.

A **shift** funcon can be specified in terms of **control** as follows:

$$\frac{E \to E'}{\textbf{shift}(E) \to \textbf{shift}(E')} \tag{24}$$

$$\frac{\mathit{val}(F) \qquad \mathit{fresh\text{-}id}(K) \qquad \mathit{fresh\text{-}id}(X)}{\begin{array}{l}\textbf{shift}(F) \to \\ \quad \textbf{control}(\textbf{lambda}(K, \textbf{apply}(F, \textbf{lambda}(X, \textbf{reset}(\textbf{apply}(\textbf{bv}(K), \textbf{bv}(X)))))))\end{array}} \tag{25}$$

The key point is the insertion of the **reset** delimiter; the rest of the lambda-term is merely an η-expansion that exposes the application of the continuation K so that the delimiter can be inserted (following [5]). Given this definition of **shift**, the **reset** delimiter coincides exactly with **prompt**:

$$\textbf{reset}(E) \to \textbf{prompt}(E) \tag{26}$$

Alternatively, the insertion of the extra delimiter could be handled by the semantics of **reset** rather than that of **shift**:

$$\frac{\mathit{val}(V)}{\textbf{reset}(V) \to V} \tag{27}$$

$$\frac{E \xrightarrow{\text{control } \textbf{none}} E'}{\textbf{reset}(E) \xrightarrow{\text{control } \textbf{none}} \textbf{reset}(E')} \tag{28}$$

$$\frac{E \xrightarrow{\text{control } \textbf{some}(F,I)} E' \qquad K = \textbf{lambda}(I, \textbf{reset}(\textbf{meta-let-in}(I, \textbf{bv}(I), E')))}{\textbf{reset}(E) \xrightarrow{\text{control } \textbf{none}} \textbf{reset}(\textbf{apply}(F, K))} \tag{29}$$

The only difference between rules 21–23 and rules 27–29 (other than the funcon names) is the definition of K in Rule 29, which here has a delimiter wrapped around the body of the continuation. Given this definition of **reset**, the **shift** operator now coincides exactly with **control**:

$$\textbf{shift}(E) \to \textbf{control}(E) \tag{30}$$

This I-MSOS specification in Rules 27–30 is similar to the evaluation-context based specification of *shift* and *reset* in [22, Section 2].

4.5 Dynamic Semantics of *abort* and *call/cc*

The *call/cc* operator is traditionally *undelimited*: it considers the current continuation to be the entirety of the rest of the program. In a setting with delimited continuations, this can be simulated by requiring there to be a single delimiter, and for it to appear at the top-level of the program. Otherwise, the two distinguishing features of *call/cc* relative to *control* and *shift* are first that an applied continuation never returns, and second that if the body of *call/cc* does not invoke a continuation, then the current continuation is applied to the result of the *call/cc* application when it returns.

To specify *call/cc*, we follow Sitaram and Felleisen [32, Section 3] and first introduce an auxiliary operator *abort*, and then specify *call/cc* in terms of *control*, *prompt* and *abort*. The purpose of *abort* is to terminate a computation (up to the innermost enclosing *prompt*) with a given value:

$$\frac{E \to E'}{\mathsf{abort}(E) \to \mathsf{abort}(E')} \tag{31}$$

$$\frac{\mathit{val}(V) \qquad \mathit{fresh\text{-}id}(I)}{\mathsf{abort}(V) \to \mathsf{control}(\mathsf{lambda}(I, V))} \tag{32}$$

The first distinguishing feature of *call/cc* is effected by placing an **abort** around any application of a continuation (preventing it from returning a value), and the second is effected by applying the continuation to the result of the F application (resuming the current continuation if F returns a value):

$$\frac{E \to E'}{\mathsf{callcc}(E) \to \mathsf{callcc}(E')} \tag{33}$$

$$\frac{\mathit{val}(F) \qquad \mathit{fresh\text{-}id}(K) \qquad \mathit{fresh\text{-}id}(X)}{\begin{array}{l}\mathsf{callcc}(F) \to \mathsf{control}(\mathsf{lambda}(K,\\ \quad \mathsf{apply}(\mathsf{bv}(K), \mathsf{apply}(F, \mathsf{lambda}(X, \mathsf{abort}(\mathsf{apply}(\mathsf{bv}(K), \mathsf{bv}(X)))))))))\end{array}} \tag{34}$$

4.6 Other Control Effects

In Section 3 we presented a direct specification of exception handling using a dedicated auxiliary entity. If **throw** and **catch** (Figure 1) were used in a program together with the control operators from this section, this would give rise to two sets of independent control effects, each with independent delimiters. An alternative would be to specify exception handling indirectly in terms of the control operators (e.g. following Sitaram and Felleisen [32]), in which case the delimiters and auxiliary entity would be shared. MSOS can specify either approach, as required by the language being specified.

Beyond the control operators discussed in this section, further and more general operators for manipulating delimited continuations exist, such as those of the CPS hierarchy [12]. These are beyond the scope of this paper, and remain an avenue for future work.

5 Related Work

A direct way to specify control operators is by giving an operational semantics based on transition rules and first-class continuations. We have taken this direct approach, though in contrast to most direct specifications of control operators (e.g. [16, 20, 21, 22, 28, 31]) our approach is based on emitting signals via labelled transitions rather than on evaluation contexts. Control operators can also be given a denotational semantics by transformation to continuation-passing style (CPS) [11, 15, 29, 31], or a lower-level operational specification by translation to abstract-machine code [6, 17]. At a higher level, algebraic characterisations of control operators have been given in terms of equational theories [16, 21].

Denotationally, any function can be rewritten to CPS by taking the continuation (itself represented as a function) as an additional argument, and applying that continuation to the value the function would have returned. A straightforward extension of this transformation [12] suffices to express *call/cc*, *shift* and *reset*; however, more sophisticated CPS transformations are needed to express *control* and *prompt* [31].

Felleisen's [17] initial specification of *control* and *prompt* used a small-step operational semantics without evaluation contexts. However, this specification otherwise differs quite significantly from ours, being based on exchange rules that push *control* outwards through the term until it encounters a *prompt*. As an exchange rule has to be defined for every other construct in the language, this approach is inherently not modular. Later specifications of *control* and *prompt* used evaluation contexts and algebraic characterisations based on the notion of *abstract continuations* [16], where continuations are represented as evaluation contexts and exchange rules are not needed. Felleisen [17] also gave a lower-level operational specification based on the CEK abstract machine, where continuations are treated as frame stacks.

The *shift* and *reset* operators were originally specified denotationally, in terms of CPS semantics [11, 12]. Continuations were treated as functions, relying on the meta-continuation approach [11] which distinguishes between outer and inner continuations. Correspondingly, the meta-continuation transformation produces abstractions that take two continuation parameters, which can be further translated to standard CPS. A big-step style operational semantics for *shift* has been given in [14]. A specification based on evaluation contexts is given in [21], together with an algebraic characterisation.

Giving a CPS semantics to *control* is significantly more complex than for *shift* [31]. This is because the continuations reified by *shift* are always delimited when applied, and so can be treated as functions, which is not the case for *control*. Different approaches to this problem have been developed, including abstract continuations [16], the monadic framework in [15], and the operational framework in [6]. Relying on the introduction of recursive continuations, [31] provides an alternative approach based on a refined CPS transform. Conversely, the difference between *control* and *shift* can manifest itself quite intuitively in the direct specification of these operators—whether in our I-MSOS specifications (Section 4.4), or in specifications using evaluation contexts [16, 21, 22, 31].

As shown in [18], *shift* can be implemented in terms of *call/cc* and mutable state, and from the point of view of expressiveness, any monad that is functionally expressible can be represented in lambda calculus with *shift* and *reset*. Moreover, *control* and *shift* are equally expressive in the untyped lambda calculus [31]. A direct implementation of *control* and *shift* has been given in [19]. A CPS-based implementation of control operators in a monadic framework is given in [15]. A semantics of *call/cc* based on an efficient implementation of evaluation contexts is provided in the K Framework [28].

6 Conclusion

We have presented a dynamic semantics for control operators in the MSOS framework, settling the question of whether MSOS is expressive enough for control operators. Our definitions are concise and modular, and do not require the use of evaluation contexts.

We have validated these semantics through a suite of 70 test programs, which we accumulated from examples in the literature on control operators ([1, 2, 4, 6, 9, 10, 11, 16, 17, 20, 31]). The language we used for testing was Caml Light, a pedagogical sublanguage of a precursor to OCaml, for which we have an existing translation to funcons from a previous case study [8]. We extended Caml Light with control operators, and specified the semantics of those operators as direct translations into the corresponding funcons presented in this paper. The generated funcon programs were then tested by our prototype funcon interpreter, which directly interprets their I-MSOS specifications. The suite of test programs, and our accompanying translator and interpreter, are available online [30].

Acknowledgments: We thank Casper Bach Poulsen, Ferdinand Vesely and the anonymous reviewers for feedback on earlier versions of this paper. We also thank Martin Churchill for his exploratory notes on adding evaluation contexts to MSOS, and Olivier Danvy for suggesting additional test programs. The reported work was supported by EPSRC grant (EP/I032495/1) to Swansea University for the PLanCompS project.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. 1, 11

[2] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Fifth Asian Symposium on Programming Languages and Sys-*

tems, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007. 11

[3] Egidio Astesiano. Inductive and operational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer, 1991. ISBN 978-3-540-53961-2. 2

[4] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2):1–39, 2005. 11

[5] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *Journal of Functional Programming*, 16(3):269–280, 2006. 7, 8

[6] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006. 10, 11

[7] Martin Churchill and Peter D. Mosses. Modular bisimulation theory for computations and values. In *16th International Conference on Foundations of Software Science and Computation Structures*, volume 7794 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2013. 1, 4

[8] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development*, volume 8989 of *Lecture Notes in Computer Science*. Springer, 2015. To appear. 1, 3, 11

[9] William Clinger. The Scheme environment: Continuations. *SIGPLAN Lisp Pointers*, 1(2):22–28, 1987. 1, 11

[10] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University, 2006. 11

[11] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, 1989. 1, 10, 11

[12] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Conference on LISP and Functional Programming*, pages 151–160. ACM, 1990. 1, 9, 10

[13] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992. 1

[14] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In *Eighth European Symposium on Programming Languages and Systems*, number 1576 in Lecture Notes in Computer Science, pages 224–242. Springer, 1999. 10

[15] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007. 10, 11

[16] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *1988 Conference on LISP and Functional Programming*, pages 52–62. ACM, 1988. 1, 2, 7, 10, 11

[17] Mattias Felleisen. The theory and practice of first-class prompts. In *15th Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988. 1, 2, 10, 11

[18] Andrzej Filinski. Representing monads. In *21st Symposium on Principles of Programming Languages*, pages 446–457. ACM, 1994. 11

[19] Martin Gasbichler and Michael Sperber. Final shift for call/cc: Direct implementation of shift and reset. In *Seventh International Conference on Functional Programming*, pages 271–282. ACM, 2002. 11

[20] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 12–23. ACM, 1995. 1, 10, 11

[21] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Eighth International Conference on Functional Programming*, pages 177–188. ACM, 2003. 10

[22] Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *9th International Symposium on Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2008. 1, 7, 8, 10

[23] Peter D. Mosses. Pragmatics of modular SOS. In *Ninth International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002. 1, 2

[24] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. 1, 2

[25] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. In *Fifth Workshop on Structural Operational Semantics*, volume 229(4) of *Electronic Notes in Theoretical Computer Science*, pages 49–66. Elsevier, 2009. 1, 2

[26] Peter D. Mosses and Ferdinand Vesely. FunKons: Component-based semantics in K. In *10th International Workshop on Rewriting Logic and Its Applications*, volume 8663 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2014. 3

[27] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981. 1, 2

[28] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. 1, 10, 11

[29] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993. 10

[30] Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses. A modular structural operational semantics for delimited continuations: Additional material, 2015. 2, 11

[31] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007. 1, 10, 11

[32] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3(1):67–99, 1990. 1, 9

Why all programmers want continuations

(but use callbacks instead)

Gabriel Kerneis

12 April 2015

Abstract: Have you ever wondered why callbacks are so pervasive in modern programming languages, and yet so hated that there is such an idiom as "callback hell"? Have you ever scratched your head so hard that you started losing your hair while debugging a maze of twisty little functions all alike? Have you ever wished you could write straightforward, linear, synchronous code, and let your programming language handle concurrency? This is 2015: why isn't your compiler able to link stack frames by itself as soon as you are writing asynchronous code? As it turns out, your compiler can in fact do this for you, and much more. It just needs a gentle push in the right direction.

This talk is a tutorial on escaping callback hell with promises and generators; examples are in Javascript, but should be accessible to any interested programmer. We first build a minimal promise implementation from first principles, discovering how the underlying hidden monad makes continuation-passing style programming easier and safer. Then, we go one step further, and throw generators into the mix to recover a direct, coroutine style, restoring sanity and reaching true enlightenment. We conclude with a brief tour of other popular programming languages, and discover that the essential building blocks are already available in most cases. Educating users about them is left as an exercise to the reader.

Acknowledgments: The author is grateful to Matt Greer and Jake Archibald for their tutorials on promises, heavily reused in this presentation. He also wishes to thank the many callback lovers (and the occasional continuation haters!) he has pitched this talk to in the last few months. Their insightful, if often fairly defensive, feedback has been the main motivation for giving this talk.

Disclaimer: The opinions expressed in this tutorial are those of the author, and do not necessarily reflect the official position of his employer. No callback has been harmed during the preparation of this tutorial.

Command injection attacks, continuations, and the Lambek calculus

Hayo Thielecke
University of Birmingham

2 March 2015

Abstract

The notes show connections between command injection attacks, continuations, and the Lambek calculus: certain command injections, such as the tautology attack on SQL, are shown to be a form of control effect that can be typed using the Lambek calculus, generalizing the double-negation typing of continuations. Lambek's syntactic calculus is a logic with two implicational connectives taking their arguments from the left and right, respectively. These connectives describe how strings interact with their left and right contexts when building up syntactic structures. The calculus is a form of propositional logic without structural rules, and so a forerunner of substructural logics like Linear Logic and Separation Logic.

Contents

List of Figures

$$
\begin{array}{rcl}
E & ::= & E + E \\
& | & n \\
& | & \texttt{return}\ n
\end{array}
$$

$$
\begin{array}{rcl}
[\![E_1 + E_2]\!] & = & \lambda k.[\![E_1]\!](\lambda x_1.[\![E_2]\!](\lambda x_2.k(x_1 + x_2)) \\
[\![n]\!] & = & \lambda k.k\,n \\
[\![\texttt{return}\ n]\!] & = & \lambda k.n
\end{array}
$$

Figure 1: Control operator `return` and its continuation semantics

1 Introduction

The aim of these notes is to draw connections between three at first sight disparate topics ranging from the practical to the theoretical side of computer science:

1. Command injection attacks;
2. continuations and control effects;
3. the Lambek calculus, a presentation of syntax as a logic or type system

Depending on the reader's background, the following may serve as an introduction to command injections, the Lambek calculus, or both. Continuations will serve as the glue between these topics, so to speak, and a basic familiarity with control operators and their typing is assumed.

We briefly recall some background on continuations. Continuations in one form or another occur in many areas of computer science, ranging from compiling to logic. Like many fundamental concepts, they have been discovered independently [21], and we may even see Gödel's work on double negation as one of the first such discoveries.

Consider an expression language with a control operator `return`, as given in Figure 1. As shown in some of the earliest work on continuation semantics [23], such a language can be given a semantics by taking a continuation as a parameter.

For example, the expression

$$(\texttt{return}\ 42) + 666$$

evaluates to 42. Intuitively, this is because the evaluation context

$$(\bigcirc + 666)$$

has been discarded by the control operator.

The typing of the continuation semantics is a generalized double negation:

$$[\![E]\!] : (\texttt{int} \to \texttt{int}) \to \texttt{int}$$

The typical presentation of continuation passing, following Plotkin [19], uses λ-calculus, but much of the machinery of continuations works in a more general situation. If fewer structural rules are assumed (omitting Contraction and Weakening), then connections with Linear Logic emerge [3, 9]. Lambek's syntactic calculus [15] goes further and removes the Exchange rule as well, making it a canonical logic for reasoning about strings. Control operators, by giving access to the current continuation, have an effect of the surrounding evaluation context. Analogously, in the Lambek calculus, a binary operator has a syntactic effect in the sense that it consumes some of its syntactic context, as given by symbols to its left and right.

2 Command injection attacks

In programming language theory, one usually assumes that all matters of parsing have been settled, so that the syntax is given as abstract syntax trees, rather than raw sequences of symbols, before language features such as types or effects are considered. However, complex software increasingly contains parsers and interpreters of various kinds, some for full programming languages, others for more restricted languages such as SQL or XML. Input to them is parsed at runtime, and may originate from untrusted sources. Consequently, syntax becomes a problem again, impacting the safety and security of the interpreters.

Command injection attacks form a large class of attacks on software (for an overview, see texts on secure programming, such as Dowd et al [6]). They may happen whenever user-malleable and potentially malicious fragments in some syntax are spliced into a syntactic context such that the resulting string is parsed and interpreted. It is crucial for the attack that the fragments to be combined are raw text that still has to be parsed, rather than some structured format such as abstract syntax trees. Of course an attacker could just inject syntactically invalid gibberish and provoke parsing errors. Depending on the robustness of error handling, that could amount to a mere nuisance or a denial of service attack. However, command injection attacks are far more pernicious by creating strings that are successfully parsed and therefore interpreted. By gaining access to the interpreter to run code of their choosing, attackers can violate integrity and confidentiality, rather than merely triggering errors.

SQL command injection [12] attacks are perhaps the best known example; in this case the constructed strings are SQL queries that are interpreted by the database management system. In some variants [12] of SQL command injection attacks, the attacker relies on injecting code with side effects, such as a `DROP` statement in SQL that destructively updates the database. In this paper we will however concentrate on a class of attacks that do not require side effects in the injected code and rely purely on *syntactically* subverting the constructed string. The so-called tautology attack is a notorious example. Simply put, a malicious

user injects the string `OR 1 = 1`, which when combined with a Boolean test renders that test tautologically true and hence useless.

Command injection attacks are by no means confined to SQL injections. They may arise whenever data is mixed with code in the broadest sense of the word; for instance, XPath injection attacks are a recent example. So even if SQL attacks are defeated by standard secure coding techniques, it is reasonable to expect that more vulnerabilities and attacks will emerge as dynamic scripting languages and XML/HTML based technologies proliferate, in which the mixing of code and data or in-band signalling that security experts warn against is a widespread risk.

At first sight, command injection attacks appear to be due to side effects in the interpreted language. Indeed, some attacks rely on the presence of effectful operations, such as inserting `UPDATE` or `DROP` in so-called "piggyback" SQL command injection attacks. The tautology attack, however, afflicts the purely functional language of Boolean expressions, and it does so by syntactic means rather than any side effects. The "essence" [24] of such attacks can be made precise in terms of parse trees. Intuitively, the programmer has an implicit understanding that the data supplied by the user should be slotted into the parse tree of the query as a leaf, below the comparison to `password`. Instead, the insertion of the operator `OR` rearranges the parse tree, so that the operator is above the test, rendering it ineffective by disjunction with the tautology. In this paper, we will focus exclusively on such syntactic attacks on purely functional languages.

A simple example of a syntactic command injection attack is known as a tautology attack. Suppose a dynamically constructed SQL query contains a comparison of the string `password` to some string supplied by the user, in order to check the user's authorization. Details of SQL syntax are not important here, but the idea of the syntactically malicious input is as follows. The query is constructed by concatenating a string ending in "`password = '`" with the user input to construct a boolean expression. This test is part of an SQL statement, as in "`SELECT * FROM table WHERE...`". If the user supplies the input "`foo`", the concatenation contains the test "`password = 'foo'`", as intended. In an attack, the user injects an operator, by supplying the input "`foo' OR 1 = 1 --`". The resulting test is

```
password = 'foo' OR 1 = 1
```

which always evaluates to true due to the tautology `1 = 1`. Using this technique, attackers may read confidential data for other users, bypass password authentication, and the like, with many variations on this theme of injecting operators [12].

Given that the phenomenon of syntactic command injection is so general, and independent of many details of the particular technologies being exploited, we aim to address it at the appropriate level of abstraction. We would like to reason about the syntactic effect, as it were, that a malicious input has on its context, similar to the way that a type and effect system [16] lets us reason

String with a hole: `password = ◯`

Legitimate input: `foo`

Combined string: `password = foo`

Malicious input: `foo OR 1 = 1`

Combined string: `password = foo OR 1 = 1`

Figure 2: Tautology attack

about side effects. The effect can be subtle, as a malicious input string needs to conform closely to the structure of the surrounding string it is intended to attack. If for instance some delimiters are added to the latter, the original attack string may fail and produce only something syntactically ill formed.

A central thesis of this paper is that the required logical tools are already available in mathematical linguistics—perhaps surprisingly so. Lambek's syntactic calculus [15] describes syntax with two (left and right) function types that capture how a phrase takes other phrases as arguments from the left or right. Using these connectives, we will explain how the harmless input "`foo`" differs from the malicious input "`foo OR 1 = 1`" that can take its place. In essence, the malicious input is effectful in that it seizes part of its context, just as a control operator does with its evaluation context. See Figure 2 (quotes are elided for simplicity).

3 Syntax and the Lambek calculus

As context-free grammars and parser generators for them are universally used in computer science, we will assume that the language we wish to reason about is given by an unambiguous context-free grammar. We will then use the Lambek calculus on top of the given grammar, not to define the language, but to describe the way its phrases combine.

We recall that a context-free grammar $\mathbf{G} = (\mathbf{T}, \mathbf{N}, \mathbf{P}, S)$ consists of a finite set $\mathbf{T}$ of terminal symbols, a finite set $\mathbf{N}$ of non-terminal symbols, a start symbol $S \in \mathbf{N}$, and a finite relation $\mathbf{P} \subseteq \mathbf{N} \times (\mathbf{N} \cup \mathbf{T})^*$ relating non-terminal symbols to strings of symbols. The elements (A, α) of $\mathbf{P}$ are called the rules or productions of the grammar, and often written as $A ::= \alpha$. (We avoid the common notation $A \to \alpha$, as it clashes with that for function types.)

We follow some notational conventions for grammars [1]. We write terminal symbols in typewriter font, as in "`a`" and "`=`". Non-terminal symbols are ranged over by A, B, C, while X and Y may be a terminal or a non-terminal symbol. Sentential forms (strings that may contain both non-terminals and terminals) are written as α, β, γ and δ. Words (strings of only terminal symbols) are ranged over by w, v, u. The empty sequence is written as ε. The one-step

derivation relation $\Rightarrow$ holds between any two strings of the form

$$\beta\, A\, \gamma \Rightarrow \beta\, \alpha\, \gamma$$

whenever there is a production $(A, \alpha) \in \mathbf{P}$. The reflexive transitive closure of $\Rightarrow$ is written as $\stackrel{*}{\Rightarrow}$.

A grammar is called unambiguous if there is no word w that has two different parse trees with root S. If we assume our grammar to be unambiguous, we are justified in speaking of "the" parse tree of a word. For a non-terminal symbol A, we say A is useless if it does not participate in the derivation of any words, that is, if there are no α, β and w such that

$$S \stackrel{*}{\Rightarrow} \alpha\, A\, \beta \stackrel{*}{\Rightarrow} w$$

We will assume that there are no useless non-terminals in the grammar (as deleting them will not change the language of the grammar). If the grammar is unambiguous and contains no useless symbols, the language of each non-terminal is also unambiguous. Unambiguous grammars are important in practice because compilers and interpreters compute meanings by induction over the parse tree; if there could be more than one such tree for a given input, there might be unintended outcomes.

When first reading about Lambek's syntactic calculus, one may perhaps be puzzled about whether to conceive of it as a form of syntax, a type system, or a logic. It is in a sense all of these, and that flexibility may be an advantage. There are two equivalent presentations of the calculus: the first as subtyping (to use current terminology), the other as a propositional logic in the style of Gentzen's sequent calculus.

Before going into the formal definitions of the calculus, it may be helpful to provide some intuition about its intended meaning, particularly compared to context-free grammars. Suppose we want to express that the operator `OR` takes a truth value T from the left and right, respectively, and produces a truth value. Using context-free grammars, we could write a grammar rule like the following:

$$T ::= T\ \texttt{OR}\ T$$

(To keep the discussion simple, let us ignore the problem of ambiguous grammars for the moment.) In the Lambek calculus, we would express the same syntactic situation differently. We would say that there is a type of of words that produce a T if a T is placed to the left of them, which we write as $T \searrow T$. Moreover, there is a type of words that produce the latter type if another T is placed to the right of them, which is written as $(T \searrow T) \swarrow T$. That gives us a type of binary operators expecting a T on either side. Stating that `OR` is such a binary operator amounts to a subtyping judgement for the type `OR` (which contains exactly the word `OR`):

$$\texttt{OR} \leq (T \searrow T) \swarrow T$$

A useful intuition to bear in mind when reading the syntactic calculus is that the left-hand side is meant to be a subset of the right-hand side. In our example

$$\frac{\varphi_1 \circ \varphi_2 \leq \psi}{\varphi_2 \leq \varphi_1 \searrow \psi} \qquad \frac{\varphi_1 \circ \varphi_2 \leq \psi}{\varphi_1 \leq \psi \swarrow \varphi_2}$$

$$\frac{\varphi_2 \leq \varphi_1 \searrow \psi}{\varphi_1 \circ \varphi_2 \leq \psi} \qquad \frac{\varphi_1 \leq \psi \swarrow \varphi_2}{\varphi_1 \circ \varphi_2 \leq \psi}$$

$$\frac{}{\varphi \leq \varphi} \qquad \frac{\varphi_1 \leq \varphi_2 \qquad \varphi_2 \leq \varphi_3}{\varphi_1 \leq \varphi_3}$$

$$\frac{}{\varphi_1 \circ (\varphi_2 \circ \varphi_3) \leq (\varphi_1 \circ \varphi_2) \circ \varphi_3}$$

$$\frac{}{(\varphi_1 \circ \varphi_2) \circ \varphi_3 \leq \varphi_1 \circ (\varphi_2 \circ \varphi_3)}$$

Figure 3: Lambek's syntactic calculus, subtyping version

here, the set containing only `OR` is a subset of the set of binary operators, but not necessarily conversely, as there may be other such operators. Note that the order of writing is reversed compared to grammar rules: a grammar rule $A ::= B$ corresponds to $B \leq A$.

If we also have `1 = 1` $\leq T$, then we see that the partial application of `OR` to it still expects a T on its left:

$$\texttt{OR 1 = 1} \leq T \searrow T$$

Thus we can construct various operators by partial application (currying), as is familiar from functional programming. It would be possible to capture the syntax of a language entirely with such judgements, without the need for a context-free grammar. However, in our setting we assume a fixed grammar is given, and we use the Lambek calculus for reasoning about fragments of words like the `OR 1 = 1` above.

We assume that a fixed context-free grammar

$$\mathbf{G} = (\mathbf{T}, \mathbf{N}, \mathbf{P}, S)$$

of interest is given, and we define a version of the syntactic calculus specific to that grammar by using its symbols as the base types and importing its rules as axioms.

$$\frac{(A, X_1 \dots X_n) \in \mathbf{P}}{X_1 \circ \dots \circ X_n \leq A}$$

$$\frac{}{\boldsymbol{\varepsilon} \circ \varphi \leq \varphi} \qquad \frac{}{\varphi \circ \boldsymbol{\varepsilon} \leq \varphi}$$

$$\frac{}{\varphi \leq \boldsymbol{\varepsilon} \circ \varphi} \qquad \frac{}{\varphi \leq \varphi \circ \boldsymbol{\varepsilon}}$$

Figure 4: Additional rules for subtyping

Definition 3.1 The types of (our variant of) the Lambek calculus are built up from the (terminal or non-terminal) symbols of our context-free grammar (ranged over by X) using the left and right arrow connectives as well as the product connective.

$$\begin{array}{rcll} \varphi, \psi, \pi & ::= & X & \text{(Grammar symbol in } \mathbf{N} \cup \mathbf{T}\text{)} \\ & | & \varphi \searrow \psi & \text{(Left implication)} \\ & | & \psi \swarrow \varphi & \text{(Right implication)} \\ & | & \varphi \circ \psi & \text{(Product)} \\ & | & \boldsymbol{\varepsilon} & \text{(Empty string type)} \end{array}$$

Definition 3.2 (Syntactic calculus, subtyping variant) The syntactic calculus consists of subtyping judgements of the form

$$\varphi \leq \psi$$

where φ and ψ are defined as in Definition 3.1. The rules for $\leq$ are given in Figures 3 and 4.

In the literature, the two implications are written as forward and backward slashes, "/" and "\". Reading such formulas can be tricky, particularly since two conventions exist. Lambek's notation places the result on top and reflects whether parameters are taken from left or right; Steedman's notation instead emphasizes the directionality of functions by placing the parameter on the left and the result on the right. We follow the Lambek style, but add arrowheads, writing "$\searrow$" and "$\swarrow$", to make it easier to see where the parameter and where the result is.

For reading nested implications, it is useful to bear in mind whether the arrows are pointing inward or outward. The following two types are isomorphic:

$$(\varphi_1 \searrow \psi) \swarrow \varphi_2 \text{ and } \varphi_1 \searrow (\psi \swarrow \varphi_2)$$

Intuitively, it makes no difference whether a binary operator consumes its left operand φ_1 or its right operand φ_2 first. We may write $\varphi_1 \searrow \psi \swarrow \varphi_2$ to mean either of them, just as brackets can be omitted due to $\circ$ being associative. By contrast, the two types where ψ occurs in a doubly negative position, as in:

$$(\varphi_1 \swarrow \psi) \searrow \varphi_2 \text{ and } \varphi_2 \swarrow (\psi \searrow \varphi_1)$$

are genuinely different, even when $\varphi_1 = \varphi_2$. Such doubly-negated types will be pertinent later on, particularly in Section 4.

The rules of the syntactic calculus are divided into those that are taken directly from Lambek's paper [15], gathered in Figure 3, and additional rules we add in this paper for using of the calculus on top of a fixed context-free grammar, presented in Figure 4.

In logical terms, the four rules for the implications in Figure 3 are quite natural if one thinks of implications (or arrow types) as adjoints of conjunctions (or products). In linear logic, the linear implication $\multimap$ is adjoint to $\varphi \otimes (-)$. In separation logic, the separating implication $-\!\!*$ is adjoint to the separating conjunction $\varphi * (-)$. In the Lambek calculus, the product is not commutative, so that $(-) \circ \varphi$ and $\varphi \circ (-)$ are not interchangeable. Consequently, there are two different adjoints $\searrow$ and $\swarrow$. The other four rules state that the subtyping relation $\leq$ is reflexive and transitive, and that the product $\circ$ is associative.

The rules in Figure 3 are the logical core of the calculus that applies to any language. In order to specialize the calculus to a particular language, we need additional axioms. In our case here, we import all productions of the given context-free grammar into the subtyping relation by adding axiom schemas stating that the product of the symbols on the right-hand side of the production is a subtype of the non-terminal symbol on the left of the production. Note that the order of the subtyping is the reverse of the way grammars are written; it is in reduction rather than derivation order. As we can have epsilon productions (having an empty string on the right-hand side) in the grammar, we need to represent the empty string ε in the syntactic calculus as well. We do so by adding a type constant called $\boldsymbol{\varepsilon}$ and rules making it a left and right unit for product. Logically, $\boldsymbol{\varepsilon}$ is a natural addition to the calculus, in that it is the nullary analogue of Lambek's binary $\circ$ connective. These rules are given in Figure 4.

Lambek [15] also defines a sequent calculus, as this yields a decision procedure. In the literature, this sequent presentation is often referred to simply as the Lambek calculus. The calculus has left and right rules for the connectives, and it lacks all structural rules, going even further than Linear Logic and Separation Logic by banishing the Exchange rule [28]. Hence it distinguishes between a left and a right implication connective.

Definition 3.3 (Sequent presentation of the calculus) Let φ, ψ and π range over types as in Definition 3.1. We let the capital Greek letters Φ, Ψ and Π range over sequences of the form $\varphi_1 \dots \varphi_n$, written without separating commas. Judgements are of the form $\Phi \triangleleft \varphi$, using the inference rules listed in Figures 5 and 6.

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \psi\ \Pi \lhd \pi}{\Psi\ (\psi \swarrow \varphi)\ \Phi\ \Pi \lhd \pi}\ (\swarrow\mathrm{L}) \qquad \frac{\Phi\ \varphi \lhd \psi}{\Phi \lhd \psi \swarrow \varphi}\ (\swarrow\mathrm{R})$$

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \psi\ \Pi \lhd \pi}{\Psi\ \Phi\ (\varphi \searrow \psi)\ \Pi \lhd \pi}\ (\searrow\mathrm{L}) \qquad \frac{\varphi\ \Phi \lhd \psi}{\Phi \lhd \varphi \searrow \psi}\ (\searrow\mathrm{R})$$

$$\frac{\Phi\ \varphi\ \psi\ \Psi \lhd \pi}{\Phi\ (\varphi \circ \psi)\ \Psi \lhd \pi}\ (\circ\mathrm{L}) \qquad \frac{\Phi \lhd \varphi \qquad \Psi \lhd \psi}{\Phi\ \Psi \lhd \varphi \circ \psi}\ (\circ\mathrm{R})$$

$$\frac{\Phi \lhd \varphi \qquad \Psi\ \varphi\ \Pi \lhd \psi}{\Psi\ \Phi\ \Pi \lhd \psi}\ (\mathrm{CUT}) \qquad \frac{}{\varphi \lhd \varphi}\ (\mathrm{AX})$$

Figure 5: Sequent calculus variant of Lambek's syntactic calculus

$$\frac{(A, \alpha) \in \mathbf{P}}{\alpha \lhd A}\ (\mathbf{P}\lhd)$$

$$\frac{\Phi\ \Psi \lhd \varphi}{\Phi\ \boldsymbol{\varepsilon}\ \Psi \lhd \varphi}\ (\boldsymbol{\varepsilon}\mathrm{L}) \qquad \frac{}{\lhd \boldsymbol{\varepsilon}}\ (\boldsymbol{\varepsilon}\mathrm{R})$$

Figure 6: Additional rules for sequents

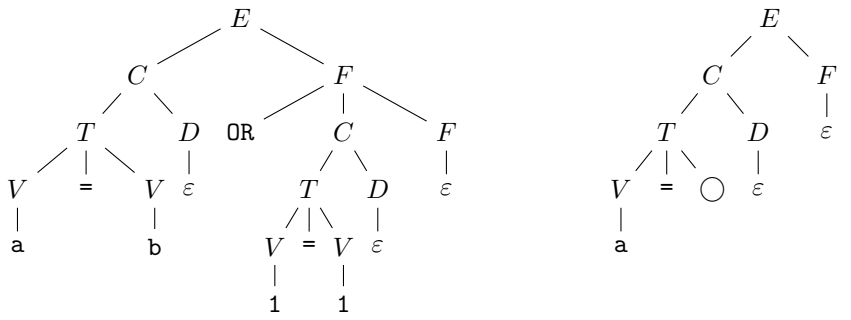


Figure 7: Parse tree for `a = b OR 1 = 1` and partial parse tree for `a =` ◯

The Lambek calculus has a simple denotational semantics. In particular, the two implications are interpreted as left and right language difference, product as concatenation, and judgements are interpreted as language inclusion [28]. For our version of the calculus, built on top of a context-free grammar, its semantics is as follows:

Definition 3.4 The denotation of a type in the syntactic calculus is a set of words, defined inductively as follows:

$$\begin{aligned}
[\![X]\!] &= \{w \in \mathbf{T}^* \mid X \stackrel{*}{\Rightarrow} w\} \\
[\![\varphi \searrow \psi]\!] &= \{w \in \mathbf{T}^* \mid \forall v \in \mathbf{T}^*.v \in [\![\varphi]\!] \text{ implies } v\,w \in [\![\psi]\!]\} \\
[\![\psi \swarrow \varphi]\!] &= \{w \in \mathbf{T}^* \mid \forall v \in \mathbf{T}^*.v \in [\![\varphi]\!] \text{ implies } w\,v \in [\![\psi]\!]\} \\
[\![\varphi_1 \circ \varphi_2]\!] &= \{w_1\,w_2 \in \mathbf{T}^* \mid w_1 \in [\![\varphi_1]\!] \text{ and } w_2 \in [\![\varphi_2]\!]\} \\
[\![\boldsymbol{\varepsilon}]\!] &= \{\varepsilon\}
\end{aligned}$$

The semantics of a logical context $\Phi = \varphi_1 \ldots \varphi_n$ is the same as that of the n-fold product of φ_j, unless the sequence is empty, in which case it is the same as $\boldsymbol{\varepsilon}$:

$$\begin{aligned}
[\![\Phi]\!] &= \{\varepsilon\} \text{ if } \Phi \text{ is the empty context} \\
[\![\varphi_1 \ \ldots \ \varphi_n]\!] &= \{w \in \mathbf{T}^* \mid w = w_1 \ldots w_n \text{ where} \\
&\qquad w_1 \in [\![\varphi_1]\!], \ldots, w_n \in [\![\varphi_n]\!]\,\}
\end{aligned}$$

4 Reasoning about syntactic effects

In this section, we first investigate a command injection attack as an example of reasoning in the syntactic calculus. Building on what can be gleaned from that example, we then place it into a wider context of types and effects.

We define a toy grammar of Boolean expressions that is sufficient for discussing tautology attacks. The grammar uses a standard technique to avoid ambiguity and to ensure that conjunction binds more tightly than disjunction [1].

An expression E is a disjunction of conjunctions C of equality tests T between values V. A series of applications of `AND` is parsed as a C, but no `OR` can appear in a C.

$$\begin{array}{rcl} E & ::= & C\ F \\ F & ::= & \texttt{OR}\ C\ F \\ & | & \varepsilon \\ C & ::= & T\ D \\ D & ::= & \texttt{AND}\ T\ D \\ & | & \varepsilon \\ T & ::= & V\ \texttt{=}\ V \\ V & ::= & 1 \mid \ldots \mid \texttt{a} \mid \texttt{b} \mid \ldots \end{array}$$

We will now reason about the interaction between malicious inputs and vulnerable contexts using the syntactic calculus in its logical variant. The presentation as a sequent calculus, with left and right rules for the connectives, may look unfamiliar compared to type systems, which are usually in natural deduction style. Nonetheless, elimination rules for the arrow types are derivable:

$$\frac{\Phi \lhd \varphi \qquad \Psi \lhd \varphi \searrow \psi}{\Phi\ \Psi \lhd \psi}\ (\searrow\text{E})$$

A symmetric elimination rule ($\swarrow$E) exists for $\swarrow$. The proof for deriving ($\searrow$E) is as follows:

$$\dfrac{\Psi \lhd \varphi \searrow \psi \qquad \dfrac{\Phi \lhd \varphi \qquad \dfrac{}{\psi \lhd \psi}\ (\text{AX})}{\Phi\ (\varphi \searrow \psi) \lhd \psi}\ (\searrow\text{L})}{\Phi\ \Psi \lhd \psi}\ (\text{CUT})$$

Now suppose we have some code in which a string variable is concatenated with the string constant "`a =` ". The judgement $\texttt{a =} \lhd T \swarrow V$ tells us that the incomplete test expects a value to its right. If we supply such a value, say `b`, we infer using the derived elimination rule:

$$\frac{\texttt{a =} \lhd T \swarrow V \qquad \texttt{b} \lhd V}{\texttt{a = b} \lhd T}\ (\swarrow\text{E})$$

Now consider the attack string `b OR 1 = 1`. The essential point is that the attack reverses the role of operator and operand when concatenated with the fragment `a =`. In our calculus that is captured by the judgement

$$\texttt{b OR 1 = 1} \lhd (T \swarrow V) \searrow E$$

To infer this, we first note that we can derive in the grammar

$$E \stackrel{*}{\Rightarrow} T\ \texttt{OR 1 = 1}$$

which implies T `OR 1 = 1` $\triangleleft E$. From that, we construct the following proof:

$$\dfrac{\dfrac{\dfrac{(V, \mathtt{b}) \in \mathbf{P}}{\mathtt{b} \triangleleft V}\,(\mathbf{P}\triangleleft) \qquad \dfrac{}{T \triangleleft T}\,(\mathrm{AX})}{(T \swarrow V)\ \mathtt{b} \triangleleft T}\,(\swarrow\mathrm{L}) \qquad T\ \mathtt{OR\ 1 = 1} \triangleleft E}{\dfrac{(T \swarrow V)\ \mathtt{b\ OR\ 1 = 1} \triangleleft E}{\mathtt{b\ OR\ 1 = 1} \triangleleft (T \swarrow V) \searrow E}\,(\searrow\mathrm{R})}\,(\mathrm{CUT})$$

The two syntax fragments fit together to build an expression:

$$\dfrac{\mathtt{a =} \triangleleft T \swarrow V \qquad \mathtt{b\ OR\ 1 = 1} \triangleleft (T \swarrow V) \searrow E}{\mathtt{a = b\ OR\ 1 = 1} \triangleleft E}\,(\searrow\mathrm{E})$$

The fragment `a =` is now in the operand position of the application, rather than the operator position it had in `a = b` $\triangleleft T$.

As Figure 7 shows, the parse tree for `a = b OR 1 = 1` does not arise from completing the partial parse tree for `a =` ◯ displayed to its right, where ◯ indicates the "hole" position in the partial parse tree and syntax fragment.

The common name in the software security literature is Tautology attack, as it is the tautology that renders the test trivially true. However, in terms of reshaping the parse tree, the crucial ingredient is the fact that the injected operator `OR` has a lower precedence that the adjacent operator `=`, as the low precedence causes the `OR` node to move up in the parse tree.

Whether or not this way of combining pieces of syntax is an attack or a useful way to build up strings depends on what type we consider the function to have.

5 Double negation in command injection and linguistics

Note that the string with the syntactic effect is very sensitive to the context on which it has an effect. If we merely change the order in the latter, replacing `a =` ◯ with ◯ `= a`, the original attack does not work anymore, producing only an ungrammatical string. (In software security practice, that means attackers may need some reverse engineering skills to craft malicious input that fits into the syntactic context like a key into a lock.) The two connectives $\searrow$ and $\swarrow$ capture such ordering accurately. For injecting into ◯ `= a`, the attack string is symmetric to the one above, with all implications reversed:

String	has type	fitting into context
`b OR 1 = 1`	$(T \swarrow V) \searrow E$	`a =` ◯
`1 = 1 OR b`	$E \swarrow (V \searrow T)$	◯ `= a`

The main use of the Lambek calculus and related formalisms such as categorial grammar has been in linguistics rather than computer science (although Lambek's original paper discusses examples from logic along with those from natural language). Nonetheless, there are some intriguing parallels to the situations we have discussed.

Consider a naive syntax for English sentences. We have a type **Sen** of sentences and a type **Noun** of nouns. Names like "`Alice`" and "`Bob`" have type **Noun**. In the syntactic calculus, a transitive verb has a type like a binary operator, for instance

$$\texttt{knows} \triangleleft \mathbf{Noun} \searrow (\mathbf{Sen} \swarrow \mathbf{Noun})$$

So we can derive sentences like "`Alice knows Bob`" in the same way as deriving Boolean expressions like `a = b OR 1 = 1`. Lambek [15] observes that pronouns like `he` and `him` may occur in some positions in which nouns may occur. However, pronouns are more sensitive to their position, because "`he`" has to occur to the left of the verb, whereas "`him`" needs to be to the right of the verb. The calculus captures this grammatical fact by giving the two different double negations as the types of "`he`" and "`him`":

String	has type	fitting into context
`he`	$\mathbf{Sen} \swarrow (\mathbf{Noun} \searrow \mathbf{Sen})$	◯ `knows Alice`
`him`	$(\mathbf{Sen} \swarrow \mathbf{Noun}) \searrow \mathbf{Sen}$	`Alice knows` ◯

Compare the difference between injection to the left or the right of the equality test discussed above.

6 Syntactic effects and control effects

Rather than supplying the expected type V, the attack string supplies a kind of generalized double negation of V, or more precisely, a V inside the negative position of two implications, as in

$$(\varphi_1 \swarrow V) \searrow \varphi_2 \text{ and } \varphi_2 \swarrow (V \searrow \varphi_1)$$

This typing generalizes the double negation of a formula A in logic, namely

$$(A \to \bot) \to \bot$$

The raising to a doubly-negated type is reminiscent of control operators in programming languages, and specifically the way that continuation passing style (CPS) introduces a form of double negation.

As a brief reminder of control operators, we consider the following simple use of the control operator `call/cc`:

$$(\texttt{call/cc}(\lambda k.42 + (k\,2))) + 1$$

Operationally, the current continuation is bound to the variable k when the call to `call/cc` is evaluated. Continuations can be represented as evaluation contexts [8], written as terms with a hole. In our example, the continuation bound to k could be written as

$$\bigcirc + 1$$

where $\bigcirc$ stand for the hole. When k is invoked in the subexpression $(k\,2)$, the value 2 is plugged into the hole of the continuation, and the whole expression thereby evaluates to $2+1 = 3$. If the operational semantics of control operators is formalized in terms of evaluation contexts [8], a salient feature is that their evaluation can move upward in the surrounding evaluation context. Compare how in Figure 7 the injected operator `OR` moves upward in the parse tree from where it was inserted by, so to speak, elbowing itself across the node labelled T.

The application $(k\,2)$ appears to be of type `int`, in that it can be used as an argument of the operator $+$. The surrounding context, expecting an integer to be supplied, can be thought of as a function from `int` to some answer type `Ans`. If a value occurs in the context, it is passed to the function, yielding an answer. However, if the expression inside the context has control effects, it does not simply supply a value to its context. Instead, it takes the context as an argument and manipulates it (in the example above, by discarding it and using the continuation bound to k instead). Hence an expression with control effects of direct-style type `int` has a continuation-passing type that is a double negation of `int`:

$$(\texttt{int} \to \texttt{Ans}) \to \texttt{Ans}$$

In programming language semantics, these double negations are inserted by continuation passing style transforms [19]. The resulting connection [11] to classical logic has been studied intensely. As a further refinement of this typing of control effects, an effect system can constrain how far up in the context the effect may reach [16, 14, 25]. In an effect system, we can control how effectful the argument of a function is. Suppose a function $f : C \to B$ is intended to be pure, which means it has no effect. The function type for pure functions is written as $C \xrightarrow{\emptyset} B$. However, if f calls a function passed as its argument, that function also needs to be pure. In the effect system, we can express this by giving a type of this form:

$$f : (A \xrightarrow{\emptyset} B) \xrightarrow{\emptyset} B$$

In an effect system, one often has a notion of sub-effecting, where a function that has fewer latent effects can be used where one with potentially more effects is expected. This fits well with out view here that a word with type φ also has the the two double negations of φ as its type, but not conversely.

To sum up, we would like to draw the following analogy between an expression with control effects and a syntactic command injection attack string:

Expression	Context expects	CPS type
$(k\ 2)$	`int`	$(\texttt{int} \to \texttt{Ans}) \to \texttt{Ans}$
`b OR 1 = 1`	V	$(T \swarrow V) \searrow E$
`1 = 1 OR b`	V	$E \swarrow (V \searrow T)$

Whereas the transformation of lambda terms into continuation passing style introduces nests of additional lambda abstractions, its analogue in the Lambek calculus is silent, so to speak. If a word w expects some word v on its right, we can regard v as expecting such a w on its left. So if v is a φ and w a $\psi \swarrow \varphi$, then we can equally regard the same word v as a $(\psi \swarrow \varphi) \searrow \psi$.

More formally, there are two derivable rules for introducing double negation:

$$\frac{\Phi \triangleleft \varphi}{\Phi \triangleleft (\psi \swarrow \varphi) \searrow \psi}\,(\textsc{dnil}) \qquad \frac{\Phi \triangleleft \varphi}{\Phi \triangleleft \psi \swarrow (\varphi \searrow \psi)}\,(\textsc{dnir})$$

These rules are derivable as follows:

$$\dfrac{\dfrac{\Phi \triangleleft \varphi \qquad \dfrac{}{\psi \triangleleft \psi}\,(\textsc{ax})}{(\psi \swarrow \varphi)\ \Phi \triangleleft \psi}\,(\swarrow\textsc{l})}{\Phi \triangleleft (\psi \swarrow \varphi) \searrow \psi}\,(\searrow\textsc{r}) \qquad \dfrac{\dfrac{\Phi \triangleleft \varphi \qquad \dfrac{}{\psi \triangleleft \psi}\,(\textsc{ax})}{\Phi\ (\varphi \searrow \psi) \triangleleft \psi}\,(\searrow\textsc{l})}{\Phi \triangleleft \psi \swarrow (\varphi \searrow \psi)}\,(\swarrow\textsc{r})$$

We recognize the syntactic control effects in the Lambek calculus as a form of continuation passing that goes even further in banishing structural rules than linear continuations [9] or linearly used continuations [3].

It is instructive to compare and contrast the two double-negation introductions in the Lambek calculus with double-negation introduction in intuitionistic and linear logic. Let us consider linear logic (as we can move from linear to intuitionistic logic by adding the Weakening and Contraction rules). There is no distinction between left and right implications, with only a single introduction and a single elimination rule for the linear implication $\multimap$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,(\multimap I) \qquad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\,(\multimap E)$$

These rules give rise to a double negation introduction. Its proof relies on the ability to exchange formulas in the context:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \multimap R \vdash A \multimap R} \qquad \Gamma \vdash A}{A \multimap R, \Gamma \vdash R}\,(\multimap E)}{\Gamma, A \multimap R \vdash R}\,(\textsc{Exchange})}{\Gamma \vdash (A \multimap R) \multimap R}\,(\multimap I)$$

The corresponding proof term is $\lambda k.k\,x$, or more precisely:

$$x : A \vdash \lambda k.k\,x : (A \multimap R) \multimap R$$

In the Lambek calculus, by contrast, there is no need for λ-abstraction and application. The continuation passing version of a word w is just w itself.

7 Conclusions

In computer science generally, the Lambek calculus, particularly when presented as sequent calculus, is perhaps chiefly recognized as an early instance of a substructural logic, dating from 1958. As such, it precedes Linear Logic [10] and the Bunched Implications [20] logic underlying Separation Logic [22]. See van Benthem's overview [28] for a comparison to Linear Logic.

It is interesting to note that the other main scourges of software security apart from command injection are memory corruption and unsafe resource usage, and that substructural logics have been successful in reasoning about memory and resource usage [13, 17, 22, 18].

Our view here of command injection as a kind of control effect that seizes its context evolved from calculi for continuations [5, 8, 7] and type-and-effect systems that make such control effects explicit in the types [11, 16, 14, 25]. Behind each continuation, it it possible to introduce another level of continuations, sometimes called meta-continuations [5]. These additional levels of continuations are particularly vivid in the syntactic calculus, as they are implicitly always present due to the silent double-negation introduction, without the need to write additional λ-abstractions. In linguistics, the double negation introduction is also known as "type raising". There are further examples of effects similar to those of control operators, such as Montague's semantics of quantification. For an introduction aimed at computer scientists, see Barker's survey article [2].

For security policies or safety properties of programming languages, there are usually dynamic (run-time) and static (compile-time) approaches. A number of tools have been developed that defend against command injection attacks in a variety of languages [24]. For such tools, a major engineering challenge is to integrate them with existing technologies such as SQL and scripting languages with minimal intervention by programmers. While the use of parsing in such defences is one of the starting points of the present paper, the focus here is much more theoretical. Thiemann's Grammar-based Analysis of String Expressions [27] uses a language of types that appears closely related to the fragment of the Lambek calculus without implications $\searrow$ and $\swarrow$.

It remains a problem for future research to establish a formal connection between syntactic effects (such as those due to command injections) and control operators in the *semantics* of the language, given by parsing actions [26]. The semantic action of a string with a syntactic effect (such as those arising in command injections) may be conjectured to be equivalent to an expression with a suitable control operator, most likely a form of delimited continuation, such as shift/reset [4].

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1985. 4, 10

[2] Chris Barker. Continuations in natural language. In *Proceedings of the Fourth ACM SIGPLAN Continuations Workshop (CW'04)*. University of Birmingham Computer Science technical report CSR-04-1, 2004. 16

[3] Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15(2/3):181–208, 2002. 2, 15

[4] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990. 16

[5] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992. 16

[6] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison Wesley, 2006. 2

[7] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages (POPL '88)*, pages 180–190. ACM, January 1988. 16

[8] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. Reasoning with continuations. In *Logic in Computer Science (LICS)*. IEEE, 1986. 14, 16

[9] Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, 1992. 2, 15

[10] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 16

[11] Timothy G. Griffin. A formulae-as-types notion of control. In *Principles of Programming Languages (POPL '90)*, pages 47–58. ACM, 1990. 14, 16

[12] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL injection attacks and countermeasures. In *ISSSE'06*. IEEE, 2006. 2, 3

[13] Samin S. Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001. 16

[14] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Programming Language Design and Implementation (PLDI)*, pages 218–226. ACM, 1988. 14, 16

[15] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958. 2, 4, 8, 13

[16] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL '88)*, pages 47–57. ACM, 1988. 3, 14, 16

[17] Gregory Morrisett, F. Smith, and D. Walker. Alias types. In *Proceedings European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000. 16

[18] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, 2004. 16

[19] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. 2, 14

[20] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Kluwer Academic Publishers, 2002. 16

[21] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993. 1

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002. 16

[23] Christopher Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974. 1

[24] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages (POPL)*, pages 372–382. ACM, 2006. 3, 16

[25] Hayo Thielecke. From control effects to typed continuation passing. In *Principles of Programming Languages (POPL'03)*, pages 139–149. ACM, 2003. 14, 16

[26] Hayo Thielecke. On the semantics of parsing actions. *Science of Computer Programming*, 84:52–76, 2014. 16

[27] Peter Thiemann. Grammar-based analysis of string expressions. In *TLDI*, pages 59–70. ACM, 2005. 16

[28] Johan van Benthem. Language in action. *Journal of Philosophical Logic*, 20(3):225–263, 1991. 8, 10, 16

A Quick Survey on Bisimulations for Delimited-Control Operators

Dariusz Biernacki Sergueï Lenglet

April 12th 2015

Abstract

We present a survey of the behavioral theory of the delimited-control operators *shift* and *reset*. We first define a notion of contextual equivalence, that we then aim to characterize with bisimilarities. We consider several styles of bisimilarities, namely normal form, applicative, and environmental. Each style has its strengths and weaknesses, and we provide several examples to allow comparisons between the different kinds of equivalence proofs.

Contents

List of Figures

1 Introduction

Control operators for delimited continuations [9, 11] provide elegant means for expressing advanced control mechanisms [9, 15]. Moreover, they play a fundamental role in the semantics of computational effects [12], normalization by evaluation [3] and as a crucial refinement of abortive control operators such as *callcc* [11, 24]. Of special interest are control operators *shift* and *reset* [9] due to their origins in continuation-passing style (CPS) and their connection with computational monads [12]. The control delimiter reset delimits the current continuation and the control operator shift abstracts the current delimited continuation as a first class value that when resumed is composed with the then-current continuation.

Because of the complex nature of control effects, it can be difficult to determine if two programs that use shift and reset are equivalent (i.e., behave in the same way) or not. *Contextual equivalence* [20] is widely considered as the most natural equivalence on terms in languages similar to the λ-calculus. The intuition behind this relation is that two programs are equivalent if replacing one by the other in a bigger program does not change the behavior of this bigger program. The behavior of a program has to be made formal by defining the *observable actions* we want to take into account for the calculus we consider. It can be, e.g., inputs and outputs for communicating systems [23], memory reads and writes, etc. For the plain λ-calculus [1], it is usually whether the term terminates or not. The "bigger program" can be seen as a *context* (a term with a hole), and therefore two terms t_0 and t_1 are contextually equivalent if we cannot tell them apart when executed within any context C, i.e., if $C[t_0]$ and $C[t_1]$ produce the same observable actions.

The latter quantification over contexts C makes context equivalence hard to use in practice to prove that two given terms are equivalent. As a result, one usually looks for more tractable alternatives to contextual equivalence, such as *bisimulations*. A bisimulation relates two terms t_0 and t_1 by asking them to mimic each other in a coinductive way, e.g., if t_0 reduces to a term t'_0, then t_1 has to reduce to a term t'_1 so that t'_0 and t'_1 are still in the bisimulation, and conversely for the reductions of t_1. An equivalence on terms, called *bisimilarity* can be derived from a notion of bisimulation: two terms are bisimilar if there exists a bisimulation which relates them. Finding an appropriate notion of bisimulation consists in finding the conditions on which two terms are related, so that the resulting notion of bisimilarity is *sound* and *complete* w.r.t. contextual equivalence, (i.e., is included into and contains contextual equivalence, respectively).

Different styles of bisimulations have been proposed for calculi similar to the λ-calculus. For example, *applicative* bisimilarity [1] relates terms by reducing them to values (if possible), and the resulting values have to be themselves applicative bisimilar when applied to an arbitrary argument. As we can see, applicative bisimilarity still contains some quantification over arguments to compare values, but is nevertheless easier to use than contextual equivalence because of its coinductive nature, and also because we do not have to consider

all forms of contexts. Applicative bisimilarity is usually sound and complete w.r.t. contextual equivalence, at least for deterministic languages such as the plain λ-calculus [1].

In contrast with applicative bisimilarity, *normal form* bisimilarity [18] does not contain any quantification over arguments or contexts in its definition. The principle is to reduce the compared terms to values (if possible), and then to decompose the resulting values into sub-components that have to be themselves bisimilar. Unlike applicative bisimilarity, normal form bisimilarity is usually not complete, i.e., there exist contextually equivalent terms that are not normal form bisimilar. But because of the lack of quantification over contexts, proving that two terms are normal form bisimilar is usually quite simple.

Finally, *environmental bisimilarity* [22] is quite similar to applicative bisimilarity, as it compares terms by reducing them to values, and then requires the resulting values to be bisimilar when applied to some arguments. However, the arguments are no longer arbitrary, but built using an *environment*, which represents the knowledge accumulated so far by an outside observer on the tested terms. Like applicative bisimilarity, environmental bisimilarity is usually sound and complete, but it also allows for up-to techniques (like normal form bisimilarity) to simplify its equivalence proofs. In contrast, the definition of up-to techniques for applicative bisimilarity remains an open problem.

In this article, we propose a survey of our previously published work [6, 5, 7] on the behavioral theory of a λ-calculus extended with the operators shift and reset. We first define a notion of contextual equivalence, that we aim to characterize with the three styles of bisimilarities discussed above. We provide several examples to show how to prove that two terms are equivalent with each bisimulation style.

Section 2 presents the syntax and semantics of the calculus $\lambda_{\mathcal{S}}$ with shift and reset we use in this paper. In this section, we also remind the definition of CPS equivalence, a CPS-based equivalence between terms, and discusse the definition of a contextual equivalence for $\lambda_{\mathcal{S}}$. We look for (at least sound) alternatives of this contextual equivalence by considering several styles of bisimilarities: normal form in Section 3, applicative in Section 4, and environmental in Section 5. Section 6 concludes this paper. Section 3 summarizes results presented in [6], Section 4 results in [5], and Section 5 results in [7].

2 The calculus $\lambda_{\mathcal{S}}$

In this section, we present the syntax, reduction semantics, and contextual equivalence for the language $\lambda_{\mathcal{S}}$ studied throughout this article.

2.1 Syntax

The language $\lambda_{\mathcal{S}}$ extends the call-by-value λ-calculus with the delimited-control operators *shift* and *reset* [9]. We assume we have a set of term variables, ranged over by x, y, z, and k. We use the metavariable k for term variables representing

a continuation (e.g., when bound with a shift), while x, y, and z stand for any values; we believe such distinction helps to understand examples and reduction rules. The syntax of terms and values is given by the following grammars:

$$\begin{array}{lrcl} \text{Terms:} & t & ::= & x \mid \lambda x.t \mid t\ t \mid \mathcal{S}k.t \mid \langle t \rangle \\ \text{Values:} & v & ::= & \lambda x.t \mid x \end{array}$$

The operator *shift* ($\mathcal{S}k.t$) is a capture operator, the extent of which is determined by the delimiter *reset* ($\langle\cdot\rangle$). A λ-abstraction $\lambda x.t$ binds x in t and a shift construct $\mathcal{S}k.t$ binds k in t; terms are equated up to α-conversion of their bound variables. The set of free variables of t is written $\mathsf{fv}(t)$; a term is *closed* if $\mathsf{fv}(t) = \emptyset$.

We distinguish several kinds of contexts, represented outside-in, as follows.

$$\begin{array}{lrcl} \text{Pure contexts:} & E & ::= & \Box \mid v\ E \mid E\ t \\ \text{Evaluation contexts:} & F & ::= & \Box \mid v\ F \mid F\ t \mid \langle F \rangle \\ \text{Contexts:} & C & ::= & \Box \mid \lambda x.C \mid t\ C \mid C\ t \mid \mathcal{S}k.C \mid \langle C \rangle \end{array}$$

Regular contexts are ranged over by C. The pure evaluation contexts[1] (abbreviated as pure contexts), ranged over by E, represent delimited continuations and can be captured by the shift operator. The call-by-value evaluation contexts, ranged over by F, represent arbitrary continuations and encode the chosen reduction strategy. Filling a context C (respectively E, F) with a term t produces a term, written $C[t]$ (respectively $E[t]$, $F[t]$); the free variables of t may be captured in the process. We extend the notion of free variables to contexts (with $\mathsf{fv}(\Box) = \emptyset$), and we say a context C (respectively E, F) is *closed* if $\mathsf{fv}(C) = \emptyset$ (respectively $\mathsf{fv}(E) = \emptyset$, $\mathsf{fv}(F) = \emptyset$). In any definitions or proofs, we say a variable is *fresh* if it does not occur free in the terms or contexts under consideration.

2.2 Reduction Semantics

The call-by-value left-to-right reduction semantics of $\lambda_{\mathcal{S}}$ is defined as follows, where $t\{v/x\}$ is the usual capture-avoiding substitution of v for x in t:

$$\begin{array}{lrcl} (\beta_v) & F[(\lambda x.t)\ v] & \to_{\mathrm{v}} & F[t\{v/x\}] \\ (\mathit{shift}) & F[\langle E[\mathcal{S}k.t] \rangle] & \to_{\mathrm{v}} & F[\langle t\{\lambda x.\langle E[x] \rangle / k\} \rangle] \text{ with } x \notin \mathsf{fv}(E) \\ (\mathit{reset}) & F[\langle v \rangle] & \to_{\mathrm{v}} & F[v] \end{array}$$

The term $(\lambda x.t)\ v$ is the usual call-by-value redex for β-reduction (rule (β_v)). The operator $\mathcal{S}k.t$ captures its surrounding context E up to the dynamically nearest enclosing reset, and substitutes $\lambda x.\langle E[x] \rangle$ for k in t (rule (*shift*)). If a reset is enclosing a value, then it has no purpose as a delimiter for a potential capture, and it can be safely removed (rule (*reset*)). All these reductions may occur within a metalevel context F. The chosen call-by-value evaluation strategy is encoded in the grammar of the evaluation contexts. Furthermore, the reduction relation $\to_{\mathrm{v}}$ is compatible with evaluation contexts F, i.e., $F[t] \to_{\mathrm{v}} F[t']$ whenever $t \to_{\mathrm{v}} t'$.

[1] This terminology comes from Kameyama (e.g., in [17]).

Example 2.1 (fixed-point combinators)**.** We remind the definition of Turing's and Curry's fixed-point combinators. Let $\theta \stackrel{\mathsf{def}}{=} \lambda xy.y\ (\lambda z.x\ x\ y\ z)$ and $\delta_x \stackrel{\mathsf{def}}{=} \lambda y.x\ (\lambda z.y\ y\ z)$; then $\Theta \stackrel{\mathsf{def}}{=} \theta\ \theta$ is Turing's call-by-value fixed-point combinator, and $\Delta \stackrel{\mathsf{def}}{=} \lambda x.\delta_x\ \delta_x$ is Curry's call-by-value fixed-point combinator. In [8], the authors propose variants of these combinators using shift and reset. They write Turing's combinator as $\langle \theta\ \mathcal{S}k.k\ k\rangle$ and Curry's combinator as $\lambda x.\langle \delta_x\ \mathcal{S}k.k\ k\rangle$. We use the combinators and their delimited-control variants as examples throghout the paper, and we study in particular the equivalences between them in Example 3.3.

There exist terms which are not values and which cannot be reduced any further; these are called *stuck terms*.

Definition 2.2. A term t is stuck if t is not a value and $t \not\to_{\mathrm{v}}$.

For example, the term $E[\mathcal{S}k.t]$ is stuck because there is no enclosing reset; the capture of E by the shift operator cannot be triggered. In fact, stuck terms are easy to characterize.

Proposition 2.3. *A term t is stuck iff $t = E[\mathcal{S}k.t']$ for some E, k, and t' or $t = F[x\ v]$ for some F, x, and v.*

We call *control stuck terms* terms of the form $E[\mathcal{S}k.t]$ and *open stuck terms* the terms of the form $F[x\ v]$.

Definition 2.4. A term t is a normal form, if t is a value or a stuck term.

We call *redexes* (ranged over by r) terms of the form $(\lambda x.t)\ v$, $\langle E[\mathcal{S}k.t]\rangle$, and $\langle v\rangle$. Thanks to the following unique-decomposition property, the reduction relation $\to_{\mathrm{v}}$ is deterministic.

Proposition 2.5. *For all terms t, either t is a normal form, or there exist a unique redex r and a unique context F such that $t = F[r]$.*

Finally, we write $\to_{\mathrm{v}}^*$ for the transitive and reflexive closure of $\to_{\mathrm{v}}$, and we define the evaluation relation of $\lambda_{\mathcal{S}}$ as follows.

Definition 2.6. We write $t \Downarrow_{\mathrm{v}} t'$ if $t \to_{\mathrm{v}}^* t'$ and t' cannot reduce further.

The result of the evaluation of a term, if it exists, is a normal form. If a term t admits an infinite reduction sequence, we say it *diverges*, written $t \Uparrow_{\mathrm{v}}$. As an example of such a term, we use extensively $\Omega \stackrel{\mathsf{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$.

In the rest of the paper, we use the following results on the reduction (or evaluation) of terms. First, a control stuck term cannot be obtained from a term of the form $\langle t\rangle$.

Proposition 2.7. *If $\langle t\rangle \Downarrow_{\mathrm{v}} t'$ then t' is a value or an open stuck term of the form $\langle F[x\ v]\rangle$. (If t is closed then t' can only be a closed value.)*

$$
\begin{array}{rcll}
(\lambda x.t)\, v & = & t\{v/x\} & \beta_{\mathtt{v}} \\
(\lambda x.E[x])\, t & = & E[t] \text{ if } x \notin \mathsf{fv}(E) & \beta_{\Omega} \\
\langle E[\mathcal{S}k.t]\rangle & = & \langle t\{\lambda x.\langle E[x]\rangle/k\}\rangle & \langle\cdot\rangle_{\mathcal{S}} \\
\langle(\lambda x.t_0)\,\langle t_1\rangle\rangle & = & (\lambda x.\langle t_0\rangle)\,\langle t_1\rangle & \langle\cdot\rangle_{\mathtt{lift}} \\
\langle v\rangle & = & v & \langle\cdot\rangle_{\mathtt{val}} \\
\mathcal{S}k.\langle t\rangle & = & \mathcal{S}k.t & \mathcal{S}_{\langle\cdot\rangle} \\
\lambda x.v\, x & = & v \text{ if } x \notin \mathsf{fv}(v) & \eta_{\mathtt{v}} \\
\mathcal{S}k.k\, t & = & t \text{ if } k \notin \mathsf{fv}(t) & \mathcal{S}_{\mathtt{elim}}
\end{array}
$$

Figure 1: Kameyama and Hasegawa's axiomatization of $\lambda_{\mathcal{S}}$

2.3 The Original Reduction Semantics

Let us notice that the reduction semantics we have introduced does not require terms to be evaluated within a top-level reset—a requirement that is commonly relaxed in practical implementations of shift and reset [10, 12], but also in some other studies of these operators [2, 16]. This is in contrast to the original reduction semantics for shift and reset [4] that has been obtained from the 2-layered continuation-passing-style (CPS) semantics [9], discussed in Section 2.4. A consequence of the correspondence with the CPS-based semantics is that terms in the original reduction semantics are treated as complete programs and are decomposed into triples consisting of a subterm (a value or a redex), a delimited context, and a meta-context (a list of delimited contexts), resembling abstract machine configurations. Such a decomposition imposes the existence of an implicit top-level reset, hard-wired in the decomposition, surrounding any term to be evaluated.

The two semantics, therefore, differ in that in the original semantics there are no stuck terms. However, it can be easily seen that operationally the difference is not essential—they are equivalent when it comes to terms of the form $\langle t\rangle$. In the rest of the article we call such terms *delimited terms* and we use the relaxed semantics when analyzing their behaviour.

The top-level reset requirement, imposed by the original semantics, does not lend itself naturally to the normal-form and applicative bismulation techniques that we propose for the relaxed semantics in Sections 3 and 4. We show, however, that the requirement can be successfully treated in the framework of environmental bisimulations, presented in Section 5.

2.4 CPS Equivalence

The operators shift and reset have been originally defined by a translation into continuation-passing style [9]. This CPS translation induces the following notion of equivalence on $\lambda_{\mathcal{S}}$ terms:

Definition 2.8. Terms t and t' are CPS equivalent if their CPS translations

are $\beta\eta$-convertible.

For example, the reduction rules $t \to_{\mathrm{v}} t'$ given in Section 2.2 are sound w.r.t. the CPS because CPS translating t and t' yields $\beta\eta$-convertible terms in the λ-calculus. The CPS equivalence has been characterized in terms of direct-style equations by Kameyama and Hasegawa who developed a sound and complete axiomatization of shift and reset [17]: two $\lambda_{\mathcal{S}}$ terms are CPS equivalent iff one can derive their equality using the equations of Figure 1.

The axiomatization is a source of examples for the bisimulation techniques that we study in Sections 3, 4 and 5, and it allows us to relate the notion of CPS equivalence to the notions of contextual equivalence that we introduce in the next section. In particular, we show that all but one axiom are validated by the bisimilarities for the relaxed semantics, and that all the axioms are validated by the environmental bisimilarity for the original semantics. The discriminating axiom that confirms the discrepancy between the two semantics is $\mathcal{S}_{\mathtt{elim}}$—the only equation that hinges on the existence of the top-level reset.

2.5 Contextual Equivalence

In this section, we discuss the possible definitions of a Morris-style contextual equivalence for the calculus $\lambda_{\mathcal{S}}$. As usual, the idea is to express that two terms are equivalent iff they cannot be distinguished when put in an arbitrary context. The question is then what kind of behavior we want to observe. In $\lambda_{\mathcal{S}}$, the evaluation of closed terms generates not only values, but also control stuck terms. Taking this into account, we obtain the following definition of contextual equivalence.

Definition 2.9. Let t_0, t_1 be closed terms. We write $t_0 \mathbb{C} t_1$ if for all closed C,

- $C[t_0] \Downarrow_{\mathrm{v}} v_0$ iff $C[t_1] \Downarrow_{\mathrm{v}} v_1$;
- $C[t_0] \Downarrow_{\mathrm{v}} t'_0$, where t'_0 is control stuck, iff $C[t_1] \Downarrow_{\mathrm{v}} t'_1$, with t'_1 control stuck as well.

The relation $\mathbb{C}$ is defined on closed terms, but can be extended to open terms using closing substitutions: we say σ closes t if it maps the free variables of t to closed values. The *open extension* of a relation, written $\mathcal{R}^\circ$, is defined as follows.

Definition 2.10. Let $\mathcal{R}$ be a relation on closed terms, and t_0 and t_1 be open terms. We write $t_0 \ \mathcal{R}^\circ \ t_1$ if for every substitution σ which closes t_0 and t_1, $t_0\sigma \ \mathcal{R} \ t_1\sigma$ holds.

The relation $\mathbb{C}$ is not suitable for the original semantics, because they distinguish terms that should be equated according to Kameyama and Hasegawa's axiomatization. Indeed, according to these relations, $\mathcal{S}k.k\, v$ (where $k \notin \mathsf{fv}(v)$) cannot be related to v (axiom $\mathcal{S}_{\mathtt{elim}}$ in Figure 1), because a stuck term cannot be related to a value. In the next section, we discuss a definition of contextual equivalence for the original semantics.

2.6 Contextual Equivalence for the Original Semantics

To reflect the fact that in the original semantics terms are evaluated within an enclosing reset, the contextual equivalence we consider for the original semantics tests terms in contexts of the form $\langle C \rangle$ only. Because delimited terms cannot reduce to stuck terms (Proposition 2.7), the only possible observable action is evaluation to values. We therefore define contextual equivalence for delimited terms as follows.

Definition 2.11. Let t_0, t_1 be closed terms. We write $t_0 \mathbb{P} t_1$ if for all closed C, $\langle C[t_0] \rangle \Downarrow_{\mathrm{v}} v_0$ iff $\langle C[t_1] \rangle \Downarrow_{\mathrm{v}} v_1$.

The relation $\mathbb{P}$ is defined on all (closed) terms, not just delimited ones. The resulting relation is less discriminative than $\mathbb{C}$, because $\mathbb{P}$ uses contexts of a particular form, while $\mathbb{C}$ tests with all contexts.

Proposition 2.12. *We have* $\mathbb{C} \subseteq \mathbb{P}$.

As a result, any equivalence between terms we prove for the relaxed semantics also holds in the original semantics, and any bisimilarity sound w.r.t. $\mathbb{C}$ (like the bisimilarities we define in Sections 3, 4, and 5.1) is also sound w.r.t. $\mathbb{P}$. However, to reach completeness, we have to design a bisimilarity suitable for delimited terms (see Section 5.2).

The inclusion of Proposition 2.12 is strict; in particular, $\mathbb{P}$ verifies the axiom $\mathcal{S}_{\mathtt{elim}}$, while $\mathbb{C}$ does not. In fact, we prove in Section 5.2 that $\mathbb{P}$ contains the CPS equivalence $\equiv$. The reverse inclusion does not hold (for $\mathbb{P}$ as well as for $\mathbb{C}$): there exists contextually equivalent terms that are not CPS equivalent.

Proposition 2.13.
1. *We have* $\Omega \mathbb{P} \Omega\Omega$ *(respectively* $\Omega \mathbb{C} \Omega\Omega$*), but* $\Omega \not\equiv \Omega\Omega$.
2. *We have* $\Theta \mathbb{P} \Delta$ *(respectively* $\Theta \mathbb{C} \Delta$*), but* $\Theta \not\equiv \Delta$.

The contextual equivalences $\mathbb{C}$ and $\mathbb{P}$ put all diverging terms in one equivalence class, while CPS equivalence is more discriminating. Furthermore, as is usual with equational theories for λ-calculi, CPS equivalence is not strong enough to equate Turing's and Curry's (call-by-value) fixed-point combinators.

As explained in the introduction, contextual equivalence is difficult to prove in practice for two given terms because of the quantification over contexts. We look for a suitable replacement (that is, an equivalence that is at least sound w.r.t. $\mathbb{C}$ or $\mathbb{P}$) by studying different styles of bisimulation in the next sections.

3 Normal Form Bisimilarity

Normal form bisimilarity [18] equates (open) terms by reducing them to normal form, and then requiring the sub-terms of these normal forms to be bisimilar. Unlike applicative and environmental bisimilarities (studied in the next sections), normal form bisimilarity usually does not contain a universal quantification over testing terms or contexts in its definition, and is therefore easier

Definition of $\star$ on values:

$$x \star y \stackrel{\mathsf{def}}{=} x\ y \qquad \lambda x.t \star y \stackrel{\mathsf{def}}{=} t\{y/x\}$$

Definition of $\mathcal{R}^{\mathsf{NF}\eta}$ on normal forms and contexts

$$\frac{E[x]\ \mathcal{R}\ E'[x] \qquad x \text{ fresh}}{E\ \mathcal{R}^{\mathsf{NF}\eta}\ E'} \qquad \frac{\langle E[x]\rangle\ \mathcal{R}\ \langle E'[x]\rangle \qquad F[x]\ \mathcal{R}\ F'[x] \qquad x \text{ fresh}}{F[\langle E\rangle]\ \mathcal{R}^{\mathsf{NF}\eta}\ F'[\langle E'\rangle]}$$

$$\frac{v_0 \star x\ \mathcal{R}\ v_1 \star x \qquad x \text{ fresh}}{v_0\ \mathcal{R}^{\mathsf{NF}\eta}\ v_1} \qquad \frac{E_0\ \mathcal{R}^{\mathsf{NF}\eta}\ E_1 \qquad \langle t_0\rangle\ \mathcal{R}\ \langle t_1\rangle}{E_0[\mathcal{S}k.t_0]\ \mathcal{R}^{\mathsf{NF}\eta}\ E_1[\mathcal{S}k.t_1]}$$

$$\frac{F_0\ \mathcal{R}^{\mathsf{NF}\eta}\ F_1 \qquad v_0\ \mathcal{R}^{\mathsf{NF}\eta}\ v_1}{F_0[x\ v_0]\ \mathcal{R}^{\mathsf{NF}\eta}\ F_1[x\ v_1]}$$

Figure 2: Definitions of the operator $\star$ and the relation $\mathcal{R}^{\mathsf{NF}\eta}$

to use than the former two. However, it is also usually not complete w.r.t. contextual equivalence, meaning that there exist contextually equivalent terms that are not normal form bisimilar. This section summarizes the results in [6].

3.1 Definition

In the λ-calculus [21, 18], the definition of normal form bisimilarity has to take into account only values and open stuck terms. In $\lambda_{\mathcal{S}}$ with the relaxed semantics, we have to relate also control stuck terms; we propose here a first way to deal with these terms, that will be refined in the next subsection. Deconstructing normal forms leads to comparing contexts as well as terms. Given a relation $\mathcal{R}$ on terms, we define in Fig. 2 an extension of $\mathcal{R}$ to normal forms and contexts, written $\mathcal{R}^{\mathsf{NF}\eta}$, which relies on an application operator for values $\star$. The rationale behind the definitions of $\star$ and $\mathcal{R}^{\mathsf{NF}\eta}$ becomes clear when we explain our notion of normal form bisimilarity, defined below.

Definition 3.1. A relation $\mathcal{R}$ on terms is a normal form simulation if $t_0\ \mathcal{R}\ t_1$ and $t_0 \Downarrow_{\mathrm{v}} t_0'$ implies $t_1 \Downarrow_{\mathrm{v}} t_1'$ and $t_0'\ \mathcal{R}^{\mathsf{NF}\eta}\ t_1'$. A relation $\mathcal{R}$ is a normal form bisimulation if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are normal form simulations. Normal form bisimilarity, written $\mathbb{N}$, is the largest normal form bisimulation.

In this section, we often drop the "normal form" attribute when it does not cause confusion. Two terms t_0 and t_1 are bisimilar if their evaluations lead to matching normal forms (e.g., if t_0 evaluates to a control stuck term, then so does t_1) with bisimilar sub-components. We now detail the different cases.

Normal form bisimilarity does not distinguish between evaluation to a variable and evaluation to a λ-abstraction. Instead, we relate terms evaluating to

any values v_0 and v_1 by comparing $v_0 \star x$ and $v_1 \star x$, where x is fresh. As originally pointed out by Lassen [18], this is necessary for the bisimilarity to be sound w.r.t. η-expansion; otherwise it would distinguish η-equivalent terms such as $\lambda y.x\, y$ and x. Using $\star$ instead of regular application avoids the introduction of unnecessary β-redexes, which could reveal themselves problematic in proofs.

For a control stuck term $E_0[\mathcal{S}k.t_0]$ to be executed, it has to be plugged into an evaluation context surrounded by a reset; by doing so, we obtain a term of the form $\langle t_0\{\lambda x.\langle E_0'[x]\rangle/k\}\rangle$ for some context E_0'. Notice that the resulting term is within a reset; similarly, when comparing $E_0[\mathcal{S}k.t_0]$ and $E_1[\mathcal{S}k.t_1]$, we ask for the shift bodies t_0 and t_1 to be related when surrounded by a reset. We also compare E_0 and E_1, which amounts to executing $E_0[x]$ and $E_1[x]$ for a fresh x, since the two contexts are pure. Comparing t_0' and t_1' without reset would be too discriminating, as it would distinguish contextually equivalent terms such as $\mathcal{S}k.\langle t\rangle$ and $\mathcal{S}k.t$ (axiom $\mathcal{S}_{\langle\cdot\rangle}$). Indeed, without reset, we would have to relate $\langle t\rangle$ and t, which are not equivalent in general (take $t = \mathcal{S}k'.v$ for some v), while Definition 3.1 requires $\langle\langle t\rangle\rangle$ and $\langle t\rangle$ to be related (which holds for all t; see Example 3.2).

The open stuck terms $F_0[x\, v_0]$ and $F_1[x\, v_1]$ are bisimilar if the values v_0 and v_1 as well as the contexts F_0 and F_1 are related. We have to be careful when defining bisimilarity on (possibly non pure) evaluation contexts. We cannot simply relate F_0 and F_1 by executing $F_0[y]$ and $F_1[y]$ for a fresh y. Such a definition would equate the contexts $\Box$ and $\langle\Box\rangle$, which in turn would relate the terms $x\, v$ and $\langle x\, v\rangle$, which are not contextually equivalent: they are distinguished by the context $(\lambda x.\Box)\, \lambda y.\mathcal{S}k.\Omega$. A context containing a reset enclosing the hole should be related only to contexts with the same property. However, we do not want to precisely count the number of delimiters around the hole; doing so would distinguish $\langle\Box\rangle$ and $\langle\langle\Box\rangle\rangle$, and therefore it would discriminate the contextually equivalent terms $\langle x\, v\rangle$ and $\langle\langle x\, v\rangle\rangle$. Hence, the definition of $\mathcal{R}^{\mathsf{NF}\eta}$ for contexts (Fig. 2) checks that if one of the contexts contains a reset surrounding the hole, then so does the other; then it compares the contexts beyond the first enclosing delimiter by simply evaluating them using a fresh variable. As a result, it rightfully distinguishes $\Box$ and $\langle\Box\rangle$, but it relates $\langle\Box\rangle$ and $\langle\langle\Box\rangle\rangle$.

We now give some examples to show how to prove equivalences using normal form bisimulation.

Example 3.2 (double reset)**.** We prove that $\langle t\rangle\ \mathbb{N}\ \langle\langle t\rangle\rangle$ by showing that $\mathcal{R} \stackrel{\mathsf{def}}{=} \{(\langle t\rangle, \langle\langle t\rangle\rangle)\} \cup \mathbb{N}$ is a bisimulation. First, note that the case $\langle t\rangle \Downarrow_{\mathrm{v}} E[\mathcal{S}k.t']$ is not possible because of Proposition 2.7. Suppose $\langle t\rangle \Downarrow_{\mathrm{v}} v$. we prove that $\langle t\rangle \Downarrow_{\mathrm{v}} v$ iff $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} v$. If $\langle t\rangle \Downarrow_{\mathrm{v}} v$, then $\langle\langle t\rangle\rangle \to_{\mathrm{v}}^* \langle v\rangle \to_{\mathrm{v}} v$. Conversely, if $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} v$, then $\langle t\rangle$ cannot diverge or cannot reduce to an open stuck term (otherwise, $\langle\langle t\rangle\rangle$ would also diverge or reduce to an open stuck term). Hence, we have $\langle t\rangle \Downarrow_{\mathrm{v}} v'$, which entails $\langle\langle t\rangle\rangle \to_{\mathrm{v}}^* \langle v'\rangle \to_{\mathrm{v}} v'$, which in turn implies $v = v'$ because normal forms are unique. Consequently, we have $\langle t\rangle \Downarrow_{\mathrm{v}} v$ iff $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} v$, and $v\ \mathbb{N}^{\mathsf{NF}\eta}\ v$ holds.

If $\langle t\rangle \Downarrow_{\mathrm{v}} F[x\, v]$, then there exists F' such that $t \Downarrow_{\mathrm{v}} F'[x\, v]$ and $F = \langle F'\rangle$.

Therefore, we have $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} \langle\langle F'[x\ v]\rangle\rangle$. We have $v\ \mathbb{N}^{\mathsf{NF}\eta}\ v$, and we have to prove that $\langle F'\rangle\ \mathcal{R}^{\mathsf{NF}\eta}\ \langle\langle F'\rangle\rangle$ to conclude. If F' is a pure context E, then we have to prove $\langle E[y]\rangle\ \mathcal{R}\ \langle E[y]\rangle$ and $y\ \mathcal{R}\ \langle y\rangle$ for a fresh y, which are both true because $\mathbb{N} \subseteq \mathcal{R}$. If $F' = F''[\langle E\rangle]$, then given a fresh y, we have to prove $\langle F''[y]\rangle\ \mathcal{R}\ \langle\langle F''[y]\rangle\rangle$ (clear by the definition of $\mathcal{R}$), and $\langle E[y]\rangle\ \mathcal{R}\ \langle E[y]\rangle$ (true because $\mathbb{N} \subseteq \mathcal{R}$).

Similarly, it is easy to check that the evaluations of $\langle\langle t\rangle\rangle$ are matched by $\langle t\rangle$.

Example 3.3 (fixed-point combinators)**.** We study here the relationships between Turing's and Curry's fixed-point combinator and their respective variants with delimited control [8] (see Example 2.1 for the definitions). First, we prove that Turing's combinator Θ is bisimilar to its variant $\Theta_{\mathcal{S}} \stackrel{\mathsf{def}}{=} \langle\theta\ \mathcal{S}k.k\ k\rangle$. We build the candidate relation $\mathcal{R}$ incrementally, starting from $(\Theta, \Theta_{\mathcal{S}})$. Evaluating these two terms, we obtain

$$\Theta \Downarrow_{\mathrm{v}} \lambda y.y\ (\lambda z.\theta\ \theta\ y\ z) \stackrel{\mathsf{def}}{=} v_0, \text{ and}$$
$$\Theta_{\mathcal{S}} \Downarrow_{\mathrm{v}} \lambda y.y\ (\lambda z.(\lambda x.\langle\theta\ x\rangle)\ (\lambda x.\langle\theta\ x\rangle)\ y\ z) \stackrel{\mathsf{def}}{=} v_1.$$

We therefore extend $\mathcal{R}$ with $(v_0 \star y, v_1 \star y)$, where y is fresh. These two new terms are open stuck, so we add there decomposition to $\mathcal{R}$. Let $v_0' \stackrel{\mathsf{def}}{=} \lambda z.\theta\ \theta\ y\ z$ and $v_1' \stackrel{\mathsf{def}}{=} \lambda z.(\lambda x.\langle\theta\ x\rangle)\ (\lambda x.\langle\theta\ x\rangle)\ y\ z$; then we add $(v_0' \star z, v_1' \star z)$ and (z, z) for a fresh z to $\mathcal{R}$. Evaluating $v_0' \star z$ and $v_1' \star z$, we obtain respectively $y\ v_0'\ z$ and $y\ v_1'\ z$; to relate these two open stuck terms, we just need to add $(x\ z, x\ z)$ (for a fresh x) to $\mathcal{R}$, since we already have $v_0'\ \mathcal{R}^{\mathsf{NF}\eta}\ v_1'$. The constructed relation $\mathcal{R}$ we obtain is a normal form bisimulation.

In contrast, Curry's combinator Δ is not bisimilar to its delimited-control variant $\Delta_{\mathcal{S}} \stackrel{\mathsf{def}}{=} \lambda x.\langle\delta_x\ \mathcal{S}k.k\ k\rangle$. Indeed, evaluating the bodies of the two values, we obtain respectively $x\ (\lambda z.\delta_x\ \delta_x\ z)$ and $\langle\langle x\ (\lambda z.(\lambda y.\langle\delta_x\ y\rangle)\ (\lambda y.\langle\delta_x\ y\rangle)\ z)\rangle\rangle$, and these open stuck terms are not bisimilar, because $\Box$ is not related to $\langle\langle\Box\rangle\rangle$ by $\mathbb{N}^{\mathsf{NF}\eta}$. In fact, Δ and $\Delta_{\mathcal{S}}$ are distinguished by the context $\Box\ \lambda x.\mathcal{S}k.\Omega$. Finally, we can prove that the two original combinators Θ and Δ are bisimilar, using the same bisimulation as in [18].

The bisimilarity $\mathbb{N}$ is sound w.r.t. contextual equivalence.

Theorem 3.4. *We have* $\mathbb{N} \subseteq \mathbb{C}$.

The following counter-example shows that the inclusion is in fact strict; normal form bisimilarity is not complete.

Proposition 3.5. *Let* $i \stackrel{\mathsf{def}}{=} \lambda y.y$. *We have* $\langle\langle x\ i\rangle\ \mathcal{S}k.i\rangle\ \mathbb{C}^{\circ}\ \langle\langle x\ i\rangle\ (\langle x\ i\rangle\ \mathcal{S}k.i)\rangle$, *but* $\langle\langle x\ i\rangle\ \mathcal{S}k.i\rangle\ \not{\mathbb{N}}\ \langle\langle x\ i\rangle\ (\langle x\ i\rangle\ \mathcal{S}k.i)\rangle$.

3.2 Proving the Axioms

We now show how the axioms can be proved using normal form bisimulation. Because we work with the relaxed semantics in this section, we remind that the

$\mathcal{S}_{\mathtt{elim}}$ axiom does not hold, as discussed in Section 2.5. The η-equivalence axiom (η_{v} axiom) holds by definition of $\mathbb{N}^{\mathsf{NF}\eta}$.

Proposition 3.6 ($\mathcal{S}_{\langle\cdot\rangle}$ axiom)**.** *We have* $\mathcal{S}k.\langle t\rangle\ \mathbb{N}\ \mathcal{S}k.t$.

Proof. We want to relate two stuck terms, so using normal form bisimulation, we have to show $\langle\langle t\rangle\rangle\ \mathbb{N}\ \langle t\rangle$ (proved in Example 3.2) and $\Box\ \mathbb{N}^{\mathsf{NF}\eta}\ \Box$ (a consequence of the fact that $\mathbb{N}$ is reflexive). □

Proposition 3.7 ($\langle\cdot\rangle_{\mathtt{lift}}$ axiom)**.** *We have* $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle\ \mathbb{N}\ (\lambda x.\langle t_0\rangle)\ \langle t_1\rangle$.

Proof. We prove that $\mathcal{R} \stackrel{\mathsf{def}}{=} \{(\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle, (\lambda x.\langle t_0\rangle)\ \langle t_1\rangle)\} \cup \{(t,t)\}$ is a normal form bisimulation. The terms $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle$ and $(\lambda x.\langle t_0\rangle)\ \langle t_1\rangle$ reduces to a normal form iff $\langle t_1\rangle$ reduces to a normal form, and according to Proposition 2.7, we have two cases.

If $\langle t_1\rangle \Downarrow_{\mathsf{v}} v$, then $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \to_{\mathsf{v}}^* \langle t_0\{v/x\}\rangle$ and $(\lambda x.\langle t_0\rangle)\langle t_1\rangle \to_{\mathsf{v}}^* \langle t_0\{v/x\}\rangle$. Therefore, $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \Downarrow_{\mathsf{v}} t''$ iff $(\lambda x.\langle t_0\rangle)\ \langle t_1\rangle \Downarrow_{\mathsf{v}} t''$, and we have $t''\ \mathcal{R}^{\mathsf{NF}\eta}\ t''$, as required.

If $\langle t_1\rangle$ reduces to an open stuck term, then $\langle t_1\rangle \Downarrow_{\mathsf{v}} \langle F[y\,v]\rangle$ by Proposition 2.7. In this case, we have $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \Downarrow_{\mathsf{v}} \langle(\lambda x.t_0)\ \langle F[y\,v]\rangle\rangle$ and also $(\lambda x.\langle t_0\rangle)\ \langle t_1\rangle \Downarrow_{\mathsf{v}} (\lambda x.\langle t_0\rangle)\ \langle F[y\,v]\rangle$. We have $\langle(\lambda x.t_0)\ \langle F\rangle\rangle\ \mathcal{R}^{\mathsf{NF}\eta}\ (\lambda x.\langle t_0\rangle)\ \langle F\rangle$ and $v\ \mathcal{R}^{\mathsf{NF}\eta}\ v$ by definition of $\mathcal{R}$, as required.

□

Proposition 3.8 (β_Ω axiom)**.** *If* $x \notin \mathsf{fv}(E)$*, then* $(\lambda x.E[x])\ t\ \mathbb{N}\ E[t]$.

Proof. We prove that $\mathcal{R} \stackrel{\mathsf{def}}{=} \{((\lambda x.E[x])\ t, E[t]), x \notin \mathsf{fv}(E)\} \cup \{(t,t)\}$ is a normal form bisimulation. If $(\lambda x.E[x])t$ evaluates to some normal form, then t evaluates to some normal form as well. We distinguish three cases. If $t \Downarrow_{\mathsf{v}} v$, then $(\lambda x.E[x])\ t \to_{\mathsf{v}}^* E[v]$ (because $x \notin \mathsf{fv}(E)$), and $E[t] \to_{\mathsf{v}}^* E[v]$. We obtain the same term in both cases, and from there, it is easy to conclude.

If $t \Downarrow_{\mathsf{v}} F[y\,v]$, then $(\lambda x.E[x])\,t \Downarrow_{\mathsf{v}} (\lambda x.E[x])\,F[y\,v]$, and $E[t] \Downarrow_{\mathsf{v}} E[F[x\,v]]$. We have to prove $v\ \mathcal{R}^{\mathsf{NF}\eta}\ v$, which is obvious, and $(\lambda x.E[x])\,F\ \mathcal{R}^{\mathsf{NF}\eta}\ E[F]$. Let z be a fresh variable. If F is a pure context E', we have to prove $(\lambda x.E[x])\ E'[z]\ \mathcal{R}\ E[E'[z]]$, which is clearly true. Otherwise $F = F'[\langle E'\rangle]$, and we have to prove $(\lambda x.E[x])\ F'[z]\ \mathcal{R}\ E[F'[z]]$, which is clearly true, and $\langle E'[z]\rangle\ \mathcal{R}\ \langle E'[z]\rangle$, which is true as well because $\mathcal{R}$ contains the identity relation.

If $t \Downarrow_{\mathsf{v}} E'[\mathcal{S}k.t']$, then we have $(\lambda x.E[x])t \Downarrow_{\mathsf{v}} (\lambda x.E[x])\,E'[\mathcal{S}k.t']$, and $E[t] \Downarrow_{\mathsf{v}} E[E'[\mathcal{S}k.t']]$. Let y be a fresh variable. We have to prove $(\lambda x.E[x])\ E'[y]\ \mathcal{R}\ E[E'[y]]$, which is clearly true, and $\langle t'\rangle\ \mathcal{R}\ \langle t'\rangle$, which is true as well.

□

4 Applicative Bisimilarity

Applicative bisimilarity has been originally defined for the lazy λ-calculus [1]. The main idea is to reduce (closed) terms to values, and then compare the resulting λ-abstractions by applying them to an arbitrary argument. In this

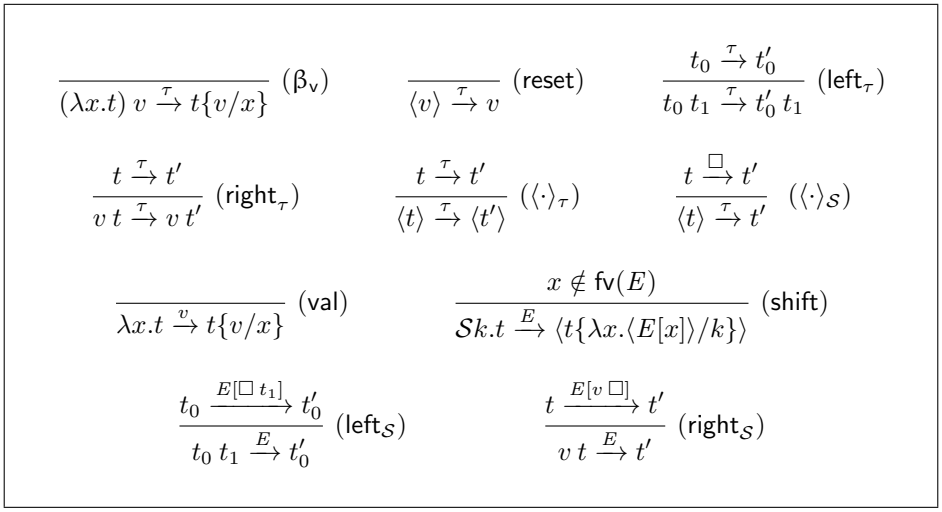


Figure 3: Labelled Transition System

section, we define a sound and complete applicative bisimilarity for the relaxed semantics of $\lambda_{\mathcal{S}}$. Our definition of applicative bisimilarity relies on a labelled transition system, introduced first. We then define the relation itself, before showing how it can be used on some examples. This section summarizes the results in [5].

4.1 Labelled Transition System

One possible way to define an applicative bisimilarity is to rely on a labelled transition system (LTS), where the possible interactions of a term with its environment are encoded in the labels (see, e.g., [14, 13]). Using a LTS simplifies the definition of the bisimilarity and makes easier to use some techniques in proofs, such as diagram chasing. In Figure 3, we define a LTS $t_0 \xrightarrow{\alpha} t_1$ with three kinds of transitions, where we assume all the terms to be closed. An *internal action* $t \xrightarrow{\tau} t'$ is an evolution from t to t' without any help from the surrounding context; it corresponds to a reduction step from t to t'. The transition $v_0 \xrightarrow{v_1} t$ expresses the fact that v_0 needs to be applied to another value v_1 to evolve, reducing to t. Finally, the transition $t \xrightarrow{E} t'$ means that t is control stuck, and when t is put in a context E enclosed in a reset, the capture can be triggered, the result of which being t'. We do not have a case for open stuck terms, because we work with closed terms only.

Most rules for internal actions (Fig. 3) are straightforward; the rules (β_v) and (reset) mimic the corresponding reduction rules, and the compositional rules (right_τ), (left_τ), and $(\langle\cdot\rangle_\tau)$ allow internal actions to happen within any evaluation context. The rule $(\langle\cdot\rangle_{\mathcal{S}})$ for context capture is explained later. Rule (val) defines the only possible transition for values. Note that while both rules (β_v) and (val) encode β-reduction, they are quite different in nature; in the former, the term

$(\lambda x.t)\ v$ can evolve by itself, without any help from the surrounding context, while the latter expresses the possibility for $\lambda x.t$ to evolve only if a value v is provided by the environment.

The rules for context capture are built following the principles of complementary semantics developed in [19]. The label of the transition $t \xrightarrow{E} t'$ contains what the environment needs to provide (a context E, but also an enclosing reset, left implicit) for the control stuck term t to reduce to t'. Hence, the transition $t \xrightarrow{E} t'$ means that we have $\langle E[t]\rangle \xrightarrow{\tau} t'$ by context capture. For example, in the rule (shift), the result of the capture of E by $\mathcal{S}k.t$ is $\langle t\{\lambda x.\langle E[x]\rangle/k\}\rangle$.

In rule ($\mathsf{left}_{\mathcal{S}}$), we want to know the result of the capture of E by the term $t_0\ t_1$, assuming t_0 contains a shift ready to perform the capture. Under this hypothesis, the capture of E by $t_0\ t_1$ comes from the capture of $E[\Box\ t_1]$ by t_0. Therefore, as premise of the rule ($\mathsf{left}_{\mathcal{S}}$), we check that t_0 is able to capture $E[\Box\, t_1]$, and the result t_0' of this transition is exactly the result we want for the capture of E by $t_0\ t_1$. The rule ($\mathsf{right}_{\mathcal{S}}$) follows the same pattern. Finally, a control stuck term t enclosed in a reset is able to perform an internal action (rule ($\langle\cdot\rangle_{\mathcal{S}}$)); we obtain the result t' of the transition $\langle t\rangle \xrightarrow{\tau} t'$ by letting t capture the empty context, i.e., by considering the transition $t \xrightarrow{\Box} t'$.

We now prove that the LTS corresponds to the reduction semantics $\rightarrow_{\mathrm{v}}$ and exhibits the observable terms (values and control stuck terms) of the language. The only difficulty is in the treatment of control stuck terms. The next lemma explicit the correspondence between $\xrightarrow{E}$ and control stuck terms.

Lemma 4.1. *If $t \xrightarrow{E} t'$, then there exist E', k, and s such that $t = E'[\mathcal{S}k.s]$ and $t' = \langle s\{\lambda x.\langle E[E'[x]]\rangle/k\}\rangle$.*

The proof is direct by induction on $t \xrightarrow{E} t'$. From this lemma, we can deduce the correspondence between $\xrightarrow{\tau}$ and $\rightarrow_{\mathrm{v}}$, and between $\xrightarrow{\alpha}$ (for $\alpha \neq \tau$) and the observable actions of the language.

Proposition 4.2. *The following hold:*

- *We have $\xrightarrow{\tau} = \rightarrow_{\mathrm{v}}$.*
- *If $t \xrightarrow{E} t'$, then t is a stuck term, and $\langle E[t]\rangle \xrightarrow{\tau} t'$.*
- *If $t \xrightarrow{v} t'$, then t is a value, and $t\ v \xrightarrow{\tau} t'$.*

4.2 Applicative Bisimilarity

We now define the notion of applicative bisimilarity for $\lambda_{\mathcal{S}}$. We write $\Rightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$. We define the weak delay[2] transition $\overset{\alpha}{\Rightarrow}$ as $\Rightarrow$ if $\alpha = \tau$ and as $\Rightarrow\xrightarrow{\alpha}$ otherwise. The definition of the (weak delay) bisimilarity is then straightforward.

[2]where internal steps are allowed before, but not after a visible action

Definition 4.3. A relation $\mathcal{R}$ on closed terms is an applicative simulation if $t_0 \mathcal{R} t_1$ implies that for all $t_0 \xrightarrow{\alpha} t_0'$, there exists t_1' such that $t_1 \stackrel{\alpha}{\Rightarrow} t_1'$ and $t_0' \mathcal{R} t_1'$. A relation $\mathcal{R}$ on closed terms is an applicative bisimulation if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations. Applicative bisimilarity $\mathbb{A}$ is the largest applicative bisimulation.

In words, two terms are equivalent if any transition from one is matched by a weak transition with the same label from the other. The relation $\mathbb{A}$ is sound and complete w.r.t. contextual equivalence.

Theorem 4.4. *We have* $\mathbb{A} = \mathbb{C}$.

We give some examples showing how applicative bisimulation can be used to prove the equivalence of terms.

Example 4.5 (double reset)**.** We show that $\langle\langle t\rangle\rangle \mathbb{A} \langle t\rangle$ holds by proving that $\mathcal{R} \stackrel{\mathsf{def}}{=} \{(\langle t\rangle, \langle\langle t\rangle\rangle)\} \cup \{(t,t)\}$ is a big-step applicative bisimulation. If $\langle t\rangle$ and/or $\langle\langle t\rangle\rangle$ is open, then $\langle t\rangle\sigma = \langle t\sigma\rangle$ (and similarly with $\langle\langle t\rangle\rangle$), for all closing substitution σ, so we still have terms in $\mathcal{R}$. With closed terms, the only possible (big-step) transition is $\langle t\rangle \stackrel{v}{\Rightarrow} t'$, which means $\langle t\rangle \Downarrow_{\mathrm{v}} v' \stackrel{v}{\rightarrow} t'$. But we have proved in Example 3.2 that $\langle t\rangle \Downarrow_{\mathrm{v}} v'$ iff $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} v'$. Consequently, we have $\langle t\rangle \stackrel{v}{\Rightarrow} t'$ iff $\langle\langle t\rangle\rangle \stackrel{v}{\Rightarrow} t'$, and we have $t' \mathcal{R} t'$, as wished. The proof is shorter than in Example 3.2 because we do not have to consider open stuck terms.

Example 4.6 (Turing's combinator)**.** We now consider Turing's combinator Θ and its variant $\Theta_{\mathcal{S}} \stackrel{\mathsf{def}}{=} \langle \theta\, \mathcal{S}k.k\, k\rangle$. The two terms can perform the following transitions.

$$\Theta \stackrel{v}{\Rightarrow} v\ (\lambda z.\theta\ \theta\ v\ z)$$
$$\Theta_{\mathcal{S}} \stackrel{v}{\Rightarrow} v\ (\lambda z.(\lambda x.\langle\theta\ x\rangle)\ (\lambda x.\langle\theta\ x\rangle)\ v\ z).$$

Assuming $v = \lambda x.t$, we have to study the behaviour of $t\{(\lambda z.\theta\ \theta\ v\ z)/x\}$, and $t\{(\lambda z.(\lambda x.\langle\theta\ x\rangle)\ (\lambda x.\langle\theta\ x\rangle)\ v\ z)/x\}$. A way to proceed is by case analysis on t, the interesting case being $t = F[x\ v']$. The resulting applicative bisimulation one can write to relate Θ and $\Theta_{\mathcal{S}}$ is much more complex than the normal form bisimulation of Example 3.3.

4.3 Proving the Axioms

As with normal form bisimulation (Section 3.2), we show how to prove Kameyama and Hasegawa's axioms (Section 2.4) except for $\mathcal{S}_{\mathtt{elim}}$ using applicative bisimulation. In the following propositions, we assume the terms to be closed, since the proofs for open terms can be deduce directly from the results with closed terms.

Proposition 4.7 (η_{v} axiom)**.** *If* $x \notin \mathsf{fv}(v)$*, then* $\lambda x.v\ x\ \mathbb{A}\ v$.

Proof. We prove that $\mathcal{R} \stackrel{\mathsf{def}}{=} \{(\lambda x.(\lambda y.t)\, x \mid \lambda y.t), x \notin \mathsf{fv}(t)\} \cup \mathbb{A}$ is a bisimulation. To this end, we have to check that $\lambda x.(\lambda y.t)x \xrightarrow{v_0} (\lambda y.t)v_0$ is matched by $\lambda y.t \xrightarrow{v_0}$

$t\{v_0/y\}$, i.e., that $(\lambda y.t)\ v_0\ \mathcal{R}\ t\{v_0/y\}$ holds for all v_0. We have $(\lambda y.t)\ v_0 \xrightarrow{\tau} t\{v_0/y\}$, and because $\xrightarrow{\tau} \subseteq \mathbb{A} \subseteq \mathcal{R}$, we have the required result. □

Proposition 4.8 ($\mathcal{S}_{\langle\cdot\rangle}$ axiom)**.** *We have* $\mathcal{S}k.\langle t\rangle\ \mathbb{A}\ \mathcal{S}k.t$.

Proof. We have $\mathcal{S}k.\langle t\rangle \xrightarrow{E} \langle\langle t\{\lambda x.\langle E[x]\rangle/k\}\rangle\rangle$ and $\mathcal{S}k.t \xrightarrow{E} \langle t\{\lambda x.\langle E[x]\rangle/k\}\rangle$ for all E. We obtain terms of the form $\langle\langle t'\rangle\rangle$ and $\langle t'\rangle$, and we have proved in Example 4.5 that $\langle\langle t'\rangle\rangle\ \mathbb{A}\ \langle t'\rangle$ holds. □

Proposition 4.9 ($\langle\cdot\rangle_{\texttt{lift}}$ axiom)**.** *We have* $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle\ \mathbb{A}\ (\lambda x.\langle t_0\rangle)\ \langle t_1\rangle$.

Proof. A transition $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \stackrel{\alpha}{\Rightarrow} t'$ (with $\alpha \neq \tau$) is possible only if $\langle t_1\rangle$ evaluates to some value v (evaluation to a control stuck terms is not possible according to Proposition 2.7). In this case, we have $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \stackrel{\tau}{\Rightarrow} \langle(\lambda x.t_0)\ v\rangle \xrightarrow{\tau} \langle t_0\{v/x\}\rangle$ and $(\lambda x.\langle t_0\rangle)\,\langle t_1\rangle \stackrel{\tau}{\Rightarrow} \langle t_0\{v/x\}\rangle$. Therefore, we have $\langle(\lambda x.t_0)\ \langle t_1\rangle\rangle \stackrel{\alpha}{\Rightarrow} t'$ (with $\alpha \neq \tau$) iff $(\lambda x.\langle t_0\rangle)\ \langle t_1\rangle \stackrel{\alpha}{\Rightarrow} t'$. From there, it is easy to conclude. □

Proposition 4.10 (β_Ω axiom)**.** *If* $x \notin \mathsf{fv}(E)$*, then* $(\lambda x.E[x])\ t\ \mathbb{A}\ E[t]$.

Sketch. We first give some intuitions on why the proof of this result is harder with applicative bisimulation than with normal form bisimulation. The difficult case is when t in the initial terms $(\lambda x.E[x])\ t$ and $E[t]$ is a control stuck term $E_0[\mathcal{S}k.t']$. Then we have the following transitions.

$$(\lambda x.E[x])\ t \xrightarrow{E_1} \langle t'\{\lambda y.\langle E_1[(\lambda x.E[x])\ E_0[y]]\rangle/k\}\rangle$$
$$E[t] \xrightarrow{E_1} \langle t'\{\lambda y.\langle E_1[E[E_0[y]]]\rangle/k\}\rangle$$

We obtain terms of the form $\langle t'\rangle\sigma$ and $\langle t'\rangle\sigma'$ (where σ and σ' are the above substitutions). We now have to consider the transitions from these terms, and the interesting case is when $\langle t'\rangle = F[k\ v]$.

$$\langle t'\rangle\sigma \xrightarrow{\tau} F\sigma[\langle E_1[(\lambda x.E[x])\ E_0[v\sigma]]\rangle] \stackrel{\mathsf{def}}{=} t_0$$
$$\langle t'\rangle\sigma' \xrightarrow{\tau} F\sigma'[\langle E_1[E[E_0[v\sigma']]]\rangle] \stackrel{\mathsf{def}}{=} t_1$$

We obtain terms that are similar to the initial terms $(\lambda x.E[x])t$ and $E[t]$, except for the extra contexts F and E_1, and the substitutions σ and σ'. Again, the interesting cases are when $E_0[v]$ is either a control stuck term, or a term of the form $F'[k\ v']$. Looking at these cases, we see that the bisimulation we have to define has to relate terms similar to t_0 and t_1, except with an arbitrary number of contexts F' and substitutions similar to σ and σ'.

□

5 Environmental Bisimilarity

Like applicative bisimilarity, environmental bisimilarity reduces closed terms to normal forms, which are then compared using some particular contexts (e.g.,

Term generating closure

$$\frac{t\ \mathcal{R}\ t'}{t\ \dot{\mathcal{R}}\ t'} \qquad x\ \dot{\mathcal{R}}\ x \qquad \frac{t\ \dot{\mathcal{R}}\ t'}{\lambda x.t\ \dot{\mathcal{R}}\ \lambda x.t'} \qquad \frac{t_0\ \dot{\mathcal{R}}\ t'_0 \qquad t_1\ \dot{\mathcal{R}}\ t'_1}{t_0\ t_1\ \dot{\mathcal{R}}\ t'_0\ t'_1} \qquad \frac{t\ \dot{\mathcal{R}}\ t'}{\mathcal{S}k.t\ \dot{\mathcal{R}}\ \mathcal{S}k.t'} \qquad \frac{t\ \dot{\mathcal{R}}\ t'}{\langle t\rangle\ \dot{\mathcal{R}}\ \langle t'\rangle}$$

Context generating closure

$$\frac{}{\Box\ \ddot{\mathcal{R}}\ \Box} \qquad \frac{F_0\ \ddot{\mathcal{R}}\ F_1 \qquad v_0\ \dot{\mathcal{R}}\ v_1}{v_0\ F_0\ \ddot{\mathcal{R}}\ v_1\ F_1} \qquad \frac{F_0\ \ddot{\mathcal{R}}\ F_1 \qquad t_0\ \dot{\mathcal{R}}\ t_1}{F_0\ t_0\ \ddot{\mathcal{R}}\ F_1\ t_1} \qquad \frac{F_0\ \ddot{\mathcal{R}}\ F_1}{\langle F_0\rangle\ \ddot{\mathcal{R}}\ \langle F_1\rangle}$$

Figure 4: Term and context generating closures

λ-abstractions are tested by passing them arguments). However, the testing contexts are not arbitrary, but built built from an environment, which represents the knowledge built so far by an outside observer. We give first the definition of environmental bisimilarity for the relaxed semantics. We then discuss a definition of environmental bisimilarity which we can prove complete for the original semantics. This section summarizes the results in [7].

5.1 Definition for the Relaxed Semantics

Environmental bisimulations use an environment $\mathcal{E}$ to accumulate knowledge about two tested terms. For the λ-calculus [22], $\mathcal{E}$ records the values (v_0, v_1) the tested terms reduce to, if they exist. We can then compare v_0 and v_1 at any time by passing them arguments built from $\mathcal{E}$. With the relaxed semantics of $\lambda_{\mathcal{S}}$, control stuck terms are also normal forms. To handle these, we allow environments to contain pairs of control stuck terms, and we test them by building pure contexts from $\mathcal{E}$. To build these testing arguments from $\mathcal{E}$, we define in Figure 4 two closures that generate respectively terms and evaluation contexts. Given a relation $\mathcal{R}$ on terms, we write $\dot{\mathcal{R}}$ for the term generating closure and $\ddot{\mathcal{R}}$ for the context generating closure. Even if $\mathcal{R}$ is defined only on closed terms, $\dot{\mathcal{R}}$ and $\ddot{\mathcal{R}}$ are defined on open terms and open contexts, respectively. In this section, we consider the restrictions of $\dot{\mathcal{R}}$ and $\ddot{\mathcal{R}}$ to respectively closed terms and closed contexts unless stated otherwise.

Formally, an environment $\mathcal{E}$ is a relation on normal forms which relates values with values and control stuck terms with control stuck terms; e.g., we define the identity environment $\mathcal{I}$ as $\{(t,t) \mid t \text{ is a normal form}\}$. An environmental relation $\mathcal{X}$ is a set of environments $\mathcal{E}$, and triples $(\mathcal{E}, t_0, t_1)$, where t_0 and t_1 are closed. We write $t_0\ \mathcal{X}_{\mathcal{E}}\ t_1$ as a shorthand for $(\mathcal{E}, t_0, t_1) \in \mathcal{X}$; roughly, it means that we test t_0 and t_1 with the knowledge $\mathcal{E}$. We define environmental bisimulation as follows.

Definition 5.1. A relation $\mathcal{X}$ is an environmental bisimulation if

1. $t_0\ \mathcal{X}_{\mathcal{E}}\ t_1$ implies:

(a) if $t_0 \to_{\mathrm{v}} t_0'$, then there exists t_1' such that $t_1 \to_{\mathrm{v}}^* t_1'$ and $t_0' \; \mathcal{X}_{\mathcal{E}} \; t_1'$;

(b) if t_0 is a normal form, then there exists a normal form t_1' of the same kind as t_0 such that $t_1 \to_{\mathrm{v}}^* t_1'$ and $\mathcal{E} \cup \{(t_0, t_1')\} \in \mathcal{X}$;

(c) the converse of the above conditions on t_1;

2. $\mathcal{E} \in \mathcal{X}$ implies:

(a) if $\lambda x.t_0 \; \mathcal{E} \; \lambda x.t_1$ and $v_0 \; \dot{\mathcal{E}} \; v_1$, then $t_0\{v_0/x\} \; \mathcal{X}_{\mathcal{E}} \; t_1\{v_1/x\}$;

(b) if $E_0[\mathcal{S}k.t_0] \; \mathcal{E} \; E_1[\mathcal{S}k.t_1]$ and $E_0' \; \ddot{\mathcal{E}} \; E_1'$, then $\langle t_0\{\lambda x.\langle E_0'[E_0[x]]\rangle/k\}\rangle \; \mathcal{X}_{\mathcal{E}} \; \langle t_1\{\lambda x.\langle E_1'[E_1[x]]\rangle/k\}\rangle$ for a fresh x.

Environmental bisimilarity, written $\approx$, is the largest environmental bisimulation. To prove that two terms t_0 and t_1 are equivalent, we want to relate them without any predefined knowledge, i.e., we want to prove that $t_0 \approx_\emptyset t_1$ holds; we also write $\mathbb{E}$ for $\approx_\emptyset$. The relation $\mathbb{E}$ will be the candidate to characterize contextual equivalence.

The first part of the definition makes the bisimulation game explicit for t_0 and t_1, while the second part focuses on environments $\mathcal{E}$. If t_0 is a normal form, then t_1 has to evaluate to a normal form of the same kind, and we extend the environment with the newly acquired knowledge. We then compare values in $\mathcal{E}$ (clause (2a)) by applying them to arguments built from $\mathcal{E}$, as in the λ-calculus [22]. Similarly, we test stuck terms in $\mathcal{E}$ by putting them within contexts $\langle E_0'\rangle$, $\langle E_1'\rangle$ built from $\mathcal{E}$ (clause (2b)) to trigger the capture. This is similar to the way we test values and stuck terms with applicative bisimilarity (Section 4), except that applicative bisimilarity tests both values or stuck terms with the same argument or context. Using different entities (as in Definition 5.1) makes bisimulation proofs harder, but it simplifies the proof of congruence of the environmental bisimilarity.

The relation we obtain is sound and complete w.r.t. contextual equivalence.

Theorem 5.2. *We have* $\mathbb{E} = \mathbb{C}$.

We now give some examples showing how the notion of environmental bisimulation can be used.

Example 5.3 (double reset)**.** We have $\langle\langle t\rangle\rangle \; \mathbb{E} \; \langle t\rangle$, because the relation

$$\{(\emptyset, \langle\langle t\rangle\rangle, \langle t\rangle)\} \cup \{(\mathcal{E}, t, t) \mid \mathcal{E} \subseteq \mathcal{I}\} \cup \{\mathcal{E} \mid \mathcal{E} \subseteq \mathcal{I}\}$$

is a big-step environmental bisimulation. Indeed, we know that $\langle\langle t\rangle\rangle \Downarrow_{\mathrm{v}} v$ iff $\langle t\rangle \Downarrow_{\mathrm{v}} v$, so we have to consider environments $\mathcal{E}$ of the form (v, v). Then, testing these $\mathcal{E}$ suppose to take $\lambda x.t \; \mathcal{E} \; \lambda x.t$ and some arguments $v_0 \; \dot{\mathcal{E}} \; v_1$, and relate $t\{v_0/x\}$ with $t\{v_1/x\}$. Since the terms related by $\mathcal{E}$ are the same, we have in fact $v_0 = v_1$, so we have to relate $t\{v_0/x\}$ with itself, hence the second set in the definition of the bisimulation.

Example 5.4 (Turing's combinator)**.** Proving that Turing's combinator Θ is bisimilar to its variant $\Theta_{\mathcal{S}} \stackrel{\mathsf{def}}{=} \langle \theta\, \mathcal{S}k.k\, k \rangle$ using the basic definition of environmental bisimulation is harder than with applicative bisimulation (Example 4.6). We remind that

$$\Theta \Downarrow_{\mathrm{v}} \lambda y.y\, (\lambda z.\theta\, \theta\, y\, z) \stackrel{\mathsf{def}}{=} v_0, \text{ and}$$
$$\Theta_{\mathcal{S}} \Downarrow_{\mathrm{v}} \lambda y.y\, (\lambda z.(\lambda x.\langle \theta\, x \rangle)\, (\lambda x.\langle \theta\, x \rangle)\, y\, z) \stackrel{\mathsf{def}}{=} v_1.$$

Therefore, we have to put (v_0, v_1) in an environment $\mathcal{E}$. When we then test v_0 and v_1, we use arguments v_0' and v_1' such that $v_0'\ \dot{\mathcal{E}}\ v_1'$, and we compare $v_0'\ (\lambda z.\theta\ \theta\ v_0'\ z)$ with $v_1'\ (\lambda z.(\lambda x.\langle \theta\, x \rangle)\ (\lambda x.\langle \theta\, x \rangle)\ v_1'\ z)$. Because we have two different terms v_0' and v_1', we can no longer do a case analysis as suggested in Example 4.6. To conclude with environmental bisimulation, we need bisimulation up to context (see [7]).

5.2 Environmental Relations for the Original Semantics

The bisimilarities introduced so far are sound and complete w.r.t. the contextual equivalence $\mathbb{C}$ of the relaxed semantics, but only sound w.r.t. the contextual equivalence $\mathbb{P}$ of the original semantics (cf. Proposition 2.12). We now propose a definition of environmental bisimulation adapted to delimited terms (but defined on all terms, like $\mathbb{P}$). Because control stuck terms cannot be obtained from the evaluation of a delimited term, environments $\mathcal{E}$ henceforth relate only values. Similarly, we write $\mathcal{R}^{\,\mathrm{v}}$ for the restriction of a relation $\mathcal{R}$ on terms to pairs of closed values.

Definition 5.5. A relation $\mathcal{X}$ is a delimited environmental bisimulation if

1. if $t_0\ \mathcal{X}_{\mathcal{E}}\ t_1$ and t_0 and t_1 are not both delimited terms, then for all closed E_0, E_1 such that $E_0\ \ddot{\mathcal{E}}\ E_1$, we have $\langle E_0[t_0] \rangle\ \mathcal{X}_{\mathcal{E}}\ \langle E_1[t_1] \rangle$;
2. $p_0\ \mathcal{X}_{\mathcal{E}}\ p_1$ implies
   (a) if $p_0 \to_{\mathrm{v}} p_0'$, then there exists p_1' such that $p_1 \to_{\mathrm{v}}^* p_1'$ and $p_0'\ \mathcal{X}_{\mathcal{E}}\ p_1'$;
   (b) if $p_0 \to_{\mathrm{v}} v_0$, then there exists v_1 such that $p_1 \to_{\mathrm{v}}^* v_1$, and $\{(v_0, v_1)\} \cup \mathcal{E} \in \mathcal{X}$;
   (c) the converse of the above conditions on p_1;
3. for all $\mathcal{E} \in \mathcal{X}$, if $\lambda x.t_0\ \mathcal{E}\ \lambda x.t_1$ and $v_0\ \dot{\mathcal{E}}\ v_1$, then $t_0\{v_0/x\}\ \mathcal{X}_{\mathcal{E}}\ t_1\{v_1/x\}$.

Delimited environmental bisimilarity, written $\simeq$, is the largest delimited environmental bisimulation. As before, the relation $\simeq_{\emptyset}$, also written $\mathbb{F}$, is candidate to characterize $\mathbb{P}$.

Clauses (2) and (3) of Definition 5.5 deal with delimited terms and environments in a classical way (as in plain λ-calculus). The problematic case is when relating terms t_0 and t_1 that are not both delimited terms (clause (1)). Indeed, one of them may be control stuck, and therefore we have to test them

within some contexts $\langle E_0 \rangle$, $\langle E_1 \rangle$ (built from $\mathcal{E}$) to potentially trigger a capture that otherwise would not happen. We cannot require both terms to be control stuck, as in clause (2b) of Definition 5.1, because a control stuck term can be equivalent to a term free from control effect. E.g., we will see that $v \mathbb{F} \mathcal{S}k.k\, v$, provided that $k \notin \mathsf{fv}(v)$.

The next proposition shows that $\mathbb{E}$ is more discriminative than $\mathbb{F}$.

Proposition 5.6. *We have* $\mathbb{E} \subseteq \mathbb{F}$.

A consequence of Proposition 5.6 is that we can use Definition 5.1 as a proof technique for $\mathbb{F}$. E.g., we have directly $\langle (\lambda x.t_0)\, \langle t_1 \rangle \rangle \mathbb{F} (\lambda x.\langle t_0 \rangle)\, \langle t_1 \rangle$, because $\langle (\lambda x.t_0)\, \langle t_1 \rangle \rangle \mathbb{E} (\lambda x.\langle t_0 \rangle)\, \langle t_1 \rangle$. The bisimilarity we obtain is sound and complete w.r.t. $\mathbb{P}$.

Theorem 5.7. *We have* $\mathbb{F} = \mathbb{P}$.

5.3 Examples

We illustrate the differences between $\mathbb{E}$ and $\mathbb{F}$, by giving some examples of terms related by $\mathbb{F}$, but not by $\mathbb{E}$. First, note that $\mathbb{F}$ relates non-terminating terms with stuck non-terminating terms.

Proposition 5.8. *We have* $\Omega \mathbb{F} \mathcal{S}k.\Omega$.

The relation $\{(\emptyset, \Omega, \mathcal{S}k.\Omega), (\emptyset, \langle E[\Omega] \rangle, \langle E[\mathcal{S}k.\Omega] \rangle), (\emptyset, \langle E[\Omega] \rangle, \langle \Omega \rangle)\}$ is a delimited bisimulation. Proposition 5.8 does not hold with $\mathbb{E}$ because Ω is not stuck.

As wished, $\mathbb{F}$ satisfies the only axiom of [17] not satisfied by $\mathbb{E}$.

Proposition 5.9. *If* $k \notin \mathsf{fv}(t)$, *then* $t \mathbb{F}^\circ \mathcal{S}k.k\, t$.

Consequently, $\mathbb{F}^\circ$ is complete w.r.t. $\equiv$.

Corollary 5.10. *We have* $\equiv \subseteq \mathbb{F}^\circ$.

As a result, we can use $\equiv$ (restricted to closed terms) as a proof technique for $\mathbb{F}$. E.g., the following equivalence can be derived from the axioms [17].

Proposition 5.11. *If* $k \notin \mathsf{fv}(t_1)$, *then* $(\lambda x.\mathcal{S}k.t_0)\, t_1 \mathbb{F} \mathcal{S}k.((\lambda x.t_0)\, t_1)$.

This equivalence does not hold with $\mathbb{E}$, because the term on the right is stuck, but the term on the left may not evaluate to a stuck term (if t_1 does not terminate).

6 Conclusion

In our study of the behavioral theory of a calculus with shift and reset, we consider two semantics: the original one, where terms are executed within an outermost reset, and the relaxed one, where this requirement is lifted. For each, we define a contextual equivalence (respectively $\mathbb{P}$ and $\mathbb{C}$), that we try to

$\curvearrowleft$		$\equiv$	$\mathbb{N}$	$\mathbb{A}$	$\mathbb{E}$	$\mathbb{F}$
relaxed semantics:	$\mathbb{C}$		$\subsetneq$	$=$	$=$	$\supsetneq$
original semantics:	$\mathbb{P}$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$\subsetneq$	$=$

Figure 5: Relationships between the equivalences of $\lambda_{\mathcal{S}}$ (e.g., $\mathbb{N} \subsetneq \mathbb{C}$)

characterize with different kinds of bisimilarities (normal form $\mathbb{N}$, applicative $\mathbb{A}$, and environmental $\mathbb{E}$, $\mathbb{F}$). We also compare our relations to CPS equivalence $\equiv$, a relation which equates terms with $\beta\eta$-equivalent CPS translations. The relationship between all these relations is summarized in Figure 5.

When comparing term equivalence proofs, we can see that each bisimulation style has its strengths and weaknesses. Normal form bisimulation arguably leads to the simplest proofs of equivalence on average, as it does not contain any quantification over arguments or testing contexts in its definition. For example, the β_Ω axiom can be easily proved using normal form bisimulation (Proposition 3.8); the proof with applicative bisimulation is much more complex (Proposition 4.10), and we do not know how to prove it with environmental bisimulation.

However, normal form bisimulation cannot be used to prove all equivalences, since its corresponding bisimilarity is not complete. It can be too discriminating to relate very simple terms, like those in Proposition 3.5. Besides, normal form bisimulation operates on open terms by definition, which requires to consider an extra normal form (open stuck terms) in the bisimulation proofs. Applicative and environmental bisimulations do not have these issues: their corresponding bisimilarities are complete, and they operate on closed terms. As a result, the proof that $\langle\langle t \rangle\rangle$ is equivalent to $\langle t \rangle$ is shorter with applicative bisimulation than with normal form bisimulation (compare Example 3.2 and Example 4.5). This is also true, e.g., for the $\langle\cdot\rangle_{\mathtt{lift}}$ axiom (compare Proposition 3.7 and 4.9).

To summarize, to prove that two given terms are equivalent, we would suggest to first try to use normal form bisimulation, and if it fails, try applicative bisimulation, and next, environmental bisimulation. This strategy holds for the relaxed as well as the original semantics, except if one wants to relate, e.g., a control stuck term with a value (like with the $\mathcal{S}_{\mathtt{elim}}$ axiom): it is possible only with the environmental bisimulation for the original semantics.

References

[1] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993. 1, 2, 11

[2] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In Zhong Shao, editor, *APLAS'07*, number 4807 in LNCS, pages 239–254, Singapore, December 2007. Springer-Verlag. 5

[3] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *POPL'04*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, January 2004. ACM Press. 1

[4] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. 5

[5] Dariusz Biernacki and Sergueï Lenglet. Applicative bisimulations for delimited-control operators. In Lars Birkedal, editor, *FOSSACS'12*, number 7213 in LNCS, pages 119–134, Tallinn, Estonia, March 2012. Springer-Verlag. 2, 12

[6] Dariusz Biernacki and Sergueï Lenglet. Normal form bisimulations for delimited-control operators. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS'12*, number 7294 in LNCS, pages 47–61, Kobe, Japan, May 2012. Springer-Verlag. 2, 8

[7] Dariusz Biernacki and Sergueï Lenglet. Environmental bisimulations for delimited-control operators. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 333–348, Melbourne, VIC, Australia, December 2013. Springer. 2, 16, 18

[8] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989. 4, 10

[9] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [25], pages 151–160. 1, 2, 5

[10] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007. 5

[11] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *POPL'88*, pages 180–190, San Diego, California, January 1988. ACM Press. 1

[12] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *POPL'94*, pages 446–457, Portland, Oregon, January 1994. ACM Press. 1, 5

[13] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, 1999. 12

[14] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming*

Languages, pages 386–395, St. Petersburg Beach, Florida, January 1996. ACM Press. 12

[15] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993. 1

[16] Yukiyoshi Kameyama. Axioms for control operators in the CPS hierarchy. *Higher-Order and Symbolic Computation*, 20(4):339–369, 2007. 5

[17] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *ICFP'03*, SIGPLAN Notices, Vol. 38, No. 9, pages 177–188, Uppsala, Sweden, August 2003. ACM Press. 3, 6, 19

[18] Søren B. Lassen. Eager normal form bisimulation. In Prakash Panangaden, editor, *LICS'05*, pages 345–354, Chicago, IL, June 2005. IEEE Computer Society Press. 2, 7, 8, 9, 10

[19] Sergueï Lenglet, Alan Schmitt, and Jean-Bernard Stefani. Howe's method for calculi with passivation. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR'09*, number 5710 in LNCS, pages 448–462, Bologna, Italy, July 2009. Springer. 13

[20] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusets Institute of Technology, 1968. 1

[21] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. In Andre Scedrov, editor, *LICS'92*, pages 102–109, Santa Cruz, California, June 1992. IEEE Computer Society. 8

[22] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 33(1):1–69, January 2011. 2, 16, 17

[23] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001. 1

[24] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [25], pages 161–175. 1

[25] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press. 21, 22

ATM without Tears: Prompt-Passing Style Transformation for Typed Delimited-Control Operators

Ikuo Kobori* Yukiyoshi Kameyama* Oleg Kiselyov†

Abstract

The salient feature of delimited-control operators is their ability to *modify* answer types during computation. The feature, answer-type modification (ATM for short), allows one to express various interesting programs such as typed printf compactly and nicely, while it makes it difficult to embed these operators in standard functional languages.

In this paper, we present a typed translation of delimited-control operators shift and reset with ATM into a familiar language with multi-prompt shift and reset without ATM, which lets us use shift and reset with ATM in standard languages without modifying the whole type system. Our translation generalizes Kiselyov's direct-style implementation of typed printf, which uses two prompts to emulate the modification of answer types. We prove that our translation preserves typing, and also give an implementation in the tagless-final style which respects typing.

*University of Tsukuba, Japan
†Tohoku University, Japan

Contents

List of Figures

1 Introduction

Delimited continuations is part of continuations, the rest of computation, and delimited-control operators provide programmers a means to access the current delimited continuations. Since the delimited-control operators control/prompt and shift/reset have been proposed around 1990 [8, 7], many researchers have been studying them intensively, to find interesting theory and application in program transformation, partial evaluation, code generation, and computational linguistics. Today, we see their implementations in many programming languages such as Scheme, Racket, SML, OCaml, Haskell, and Scala.

However, there still exists a big gap between theory and practice if we work in typed languages. Theoretically, the salient feature of delimited-control operators is their ability to modify answer types. The term `reset (3 + shift k -> k)` looks as if it has type `int`, but the result of this computation is a continuation `fun x -> reset (3 + x)` whose type is `int -> int`, which means that the initial answer type has been modified during the computation of the shift term. While this feature, called Answer-Type Modification, allows one to express surprisingly interesting programs such as typed printf, it is the source of the problem that we cannot embed the delimited-control operators in standard languages. We can hardly expect that the whole type system of a full-fledged language would be modified in such a way. With a few exceptions of Scala [12] and OchaCaml [11], we cannot directly express the beautiful examples with ATM as programs in standard languages.

This paper addresses this problem, and presents a solution for it. We will give a translation from the language with ATM shift and reset into another language with multi-prompt shift and reset without ATM. The translation is a generalization of Kiselyov's implementation of typed printf, which introduces two prompts (tags for control operators) for the answer types before and after the computation. The resulting term passes prompts of control operators during computation, and following Continuation-Passing Style, we call is Prompt-Passing Style (PPS).

The rest of this paper is organized as follows: Section 2 explains delimited-control operators and answer-type modification by a simple example. Section 3 informally states how we simulate answer-type modification using multi-prompt shift and reset, and Section 4 gives a formal account to it including formal properties. Section 5 describes the syntax-directed translation and its property. Based on the theoretical development, Section 6 gives a tagless-final implementation of shift and reset with answer-type modification as well as several programming examples. Section 7 gives related work and concluding remarks.

2 Delimited-Control Operators and Answer-type Modification

We introduce a simple example which uses delimited-control operators shift and reset where the answer types are modified through computation.

The following implementation of the `append` function is taken from Asai and Kameyama's paper [1].

```
let rec append lst = match lst with
  | [] -> shift (fun k -> k)
  | x :: xs -> x :: append xs
in let append123 =
  reset (append [1;2;3])
in
  append123 [4;5;6]
```

The function `append` takes a value of type `int list` as its input, and traverses the list. When it reaches at the end of the list, it captures the continuation (`fun ys -> reset 1 :: 2 :: 3 :: ys` in the functional form) up to the nearest reset, and returns the continuation as its result. We then apply it to the list `[4;5;6]` to obtain `[1;2;3;4;5;6]`, and it is easy to see that the function deserves its name.

Let us check the type of `append`. At the beginning, the return type of `append` (called its answer type) is `int list`, since in the second branch of the case analysis, it returns `x :: append xs`. However, the final result is a function from list to list, which is different from our initial guess. The answer type has been modified during the execution of the program.

Since its discovery, this feature has been used in many interesting examples with shift and reset, from typed printf to suspended computations, to coroutines, and even to computational linguistics. Nowadays, it is considered as one of the most attractive features of shift and reset.

Although the feature, answer-type modification, is interesting and sometimes useful, it is very hard to directly embed such control operators in conventional functional programming languages such as OCaml, as it requires a big change of the type system; a typing judgment in the form $\Gamma \vdash e : \tau$ must be changed to a more complex form $\Gamma \vdash e : \tau; \alpha, \beta$ where α and β designate the answer types before and after the execution of e. Although adjusting a type system in this way is straightforward in theory, it is rather difficult to modify existing implementations of type systems, and we therefore need a way to represent the above features in terms of standard features and/or mild extensions of existing programming languages.

This paper addresses this problem, and proposes a way to translate away the feature of ATM using multi-prompt control operators.

3 Simulating ATM with Multi-prompt shift/reset

In this section, we explain the basic ideas of our translation. Kiselyov implemented typed printf in terms of shift and reset without ATM, and we have generalized it to a translation from arbitrary terms in the source language.

Consider a simple example with answer-type modification: $[\![\langle 5 + \mathcal{S}k.k\rangle]\!]$ in which $\mathcal{S}$ is the delimited-control operator shift, and $\langle\cdots\rangle$ is reset. Its answer type changes through computation, as its initial answer type is `int` while its final answer type is `int->int`.

Let us translate the example [1] where $[\![e]\!]$ denotes the result of the translation of the term e.

We begin with the translation of a reset expression:

$$[\![\langle e\rangle]\!] = \mathcal{P}p.\mathcal{P}q.\langle \text{let } y = [\![e]\!]pq \text{ in } \mathcal{S}_q z.\ y\rangle_p$$

where the primitive $\mathcal{P}p$ creates a new prompt and binds the variable p to it. For brevity, the variable p which stores a prompt may also be called a prompt.

The translated term, when it is executed, first creates new prompts p and q and its body e is applied to the arguments p and q. Its result is stored in y and then we execute $\mathcal{S}_q z.y$, but there is no reset with the prompt q around it. Is it an error ? Actually, no. As we will see the definition below, $[\![e]\!]$ is always in the form $\lambda p.\lambda q.e'$ and during the computation of e', $\mathcal{S}_p$ is *always* invoked. Hence e' never returns normally, and the "no-reset" error does not happen. Our invariants in the translation are that the first argument (the prompt p) corresponds to the reset surrounding the expression being translated, and the second argument (the prompt q) corresponds to the above (seemingly dangerous) shift.

From the viewpoint of typing, for each occurrence of answer-type modification from α to β, we use two prompts to simulate the behavior. The prompts p and q generated here correspond to the answer types α and β, respectively.

We translate the term 5 to $[\![5]\!] = \lambda p.\lambda q.\ \mathcal{S}_p k.\ \langle k\ 5\rangle_q$ and the term $\langle 5\rangle$ is translated (essentially) to:

$$\mathcal{P}p.\mathcal{P}q.\langle \text{let } y = \mathcal{S}_p k.\langle k\ 5\rangle_q \text{ in } \mathcal{S}_q z.\ y\rangle_p$$

When we execute the result, $\mathcal{S}_p$ captures its surrounding evaluation context $\langle \text{let } y = [\ ] \text{ in } \mathcal{S}_q z.\ y\rangle_p$, binds k to its functional form $\lambda x.\langle \text{let } y = x \text{ in } \mathcal{S}_q z.\ y\rangle_p$, and continues the evaluation of $\langle k\ 5\rangle_q$. Then we get:

$$\langle\langle\langle \text{let } y = 5 \text{ in } \mathcal{S}_q z.\ y\rangle_p\rangle_q\rangle_p$$

and when this $\mathcal{S}_q$ is invoked, it is surrounded by a reset with the prompt q, and thus it is *safe*. The final result of this computation is 5. In this case, since the execution of the term 5 does not modify the answer type, the prompts p and q passed to the term $[\![5]\!]$ correspond to the same answer type, but we will soon see an example in which they correspond to different answer types.

A shift-expression is translated to:

$$[\![\mathcal{S}k.e]\!] = \lambda p.\lambda q.\mathcal{S}_p k'.\text{let } k = (\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q) \text{ in } (\!|e|\!)$$

As we have explained, p is the prompt for the reset surrounding this expression, hence $\mathcal{S}_p$ in the translated term will capture a delimited continuation up to the

[1] The precise definition of the translation is given later.

reset (which, in turn, corresponds to the nearest reset in the source term). However the delimited continuation contains a dangerous shift at its top position, so we must somehow detoxify it. For this purpose, we replace the captured continuation k' by a function $\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q$ in which the calls to k' is enclosed by a reset with the prompt q, and the dangerous shift in k' will be surrounded by it, sanitizing the dangerous behavior.

Let us consider the types of captured continuations in this translation. Suppose the term $\mathcal{S}k.e$ modifies the answer type from α to β. We use the prompts p and q, whose answer types[2] are β and α, respectively. In the source term, the continuation captured by shift (and then bound to k) has the type $\tau \to \alpha$. In the translated term, the continuation bound to k' has the type $\tau \to \beta$, since the continuation was captured by a shift with the prompt p. After some calculation, it can be inferred that the term $\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q$ has the type $\tau \to \alpha$, hence we can substitute it for k. [3]

We show the mechanism for detoxifying a dangerous shift by executing $[\![\langle 5 + \mathcal{S}k.k\rangle]\!]$, which is equivalent to:

$$\mathcal{P}p.\mathcal{P}q.\langle \text{let } y = \mathcal{P}r.((\mathcal{S}_r k.\langle k\, 5\rangle_q) + (\mathcal{S}_p k'.\text{let } k = \lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r \text{ in } k)) \text{ in } \mathcal{S}_q z.y\rangle_p$$

where the subterm starting with $\mathcal{S}_r$ is the translation result of 5, and the one with $\mathcal{S}_p$ is that of $\mathcal{S}k.k$. In general, each subterm may modify answer types. Hence, a term $e_1 + e_2$ needs three prompts corresponding to the initial, final, and intermediate answer types. The prompt r generated here corresponds to the intermediate answer type.

Evaluating this term in call-by-value, and right-to-left order (after generating all the prompts) leads to the term: $\langle \text{let } k = \lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r \text{ in } k\rangle_p$ where k' is the delimited continuation $\lambda x.\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + x \text{ in } \mathcal{S}_q z.y\rangle_p$. The result of this computation is $\lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r$, which is essentially equivalent to $\lambda y.\langle 5 + y\rangle$. To see this, applying it to 9 yields:

$$\begin{aligned} &(\lambda u.\langle(\lambda w.\Omega)((\lambda x.\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + x \text{ in } \mathcal{S}_q z.y\rangle_p)u)\rangle_r)\, 9 \\ \rightsquigarrow^* &\langle(\lambda w.\Omega)\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r \end{aligned}$$

$\mathcal{S}_r k.\langle k\, 5\rangle_q$ captures the context with the dangerous shift

$$\begin{aligned} &\rightsquigarrow^* \langle\langle(\lambda u.\langle(\lambda w.\Omega)\langle \text{let } y = u + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r)5\rangle_q\rangle_r \\ &\rightsquigarrow^* \langle\langle\langle(\lambda w.\Omega)\langle \text{let } y = 5 + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r\rangle_q\rangle_r \\ &\rightsquigarrow^* \langle\langle 14\rangle_q\rangle_r \quad \text{which reduces to 14.} \end{aligned}$$

Thus, our translation uses two prompts to make connections to two answer types, where prompts are generated dynamically.

[2] We assume that, our target language after the translation has multi-prompt shift and reset, but no answer-type modification. Hence, each prompt has a unique answer type.

[3] Here Ω is a term which has an arbitrary type. Such a term can be expressed, as, for instance, $\mathcal{P}p.\mathcal{S}_p k.\lambda x.x$. Its operational behavior does not matter, as it will be never executed.

$$
\begin{array}{rrcl}
\text{(evaluation contexts)} & E & ::= & [\,] \mid eE \mid Ev \mid \langle E \rangle \\
\text{(pure evaluation contexts)} & F & ::= & [\,] \mid eF \mid Fv
\end{array}
$$

$$
\begin{array}{rll}
E[(\lambda x.e)\, v] & \rightsquigarrow E[e\{v/x\}] & \\
E[\text{let } x = v \text{ in } e] & \rightsquigarrow E[e\{v/x\}] & \\
E[\langle v \rangle] & \rightsquigarrow E[v] & \\
E[\langle F[\mathcal{S}k.e] \rangle] & \rightsquigarrow E[\langle e\{\lambda y.\langle F[y] \rangle / k\} \rangle] & y \text{ is a fresh variable in } F
\end{array}
$$

Figure 1: Operational Semantics of Source Calculus

4 Source and Target Calculi

In this section, we formally define our source and target calculi.

The source calculus is based on Asai and Kameyama's polymorphic extension of Danvy and Filinski's calculus for shift and reset, both of which allow answer-type modification [6, 1]. We slightly modified it here; (1) we removed fixpoint and conditionals (but they can be added easily), (2) we use value restriction for let-polymorphism while they used more relaxed condition, and (3) we use Biernacki et al.'s simplification for the types of delimited continuations [4].

The syntax of values and terms of our source calculus λ^{ATM} is defined as follows:

$$
\begin{array}{rrcl}
\text{(values)} & v & ::= & x \mid c \mid \lambda x.e \\
\text{(terms)} & e & ::= & v \mid e_1 e_2 \mid \text{let } x = v \text{ in } e \mid \mathcal{S}k.e \mid \langle e \rangle
\end{array}
$$

where $\lambda x.e$ and $\mathcal{S}k.e$ bind x and k in e, resp.

Figure 1 defines call-by-value operational semantics to the language above.

The term $[\,]$ denotes the empty context. Evaluation contexts are standard, and pure evaluation contexts are those evaluation contexts that have no resets enclosing the hole. Note that we use the right-to-left evaluation order for the function applications to reflect the current OCaml compiler's semantics.

The first two evaluation rules are the standard beta and let rules, where $e\{v/x\}$ denotes capture-avoiding substitution. The next two rules are those for control operators: if the body of a reset expression is a value, the occurrence of reset is discarded. If the next redex is a shift expression, we capture the continuation up to the nearest reset and bind k to it.

Figure 2 introduces types and related notions. Types are type variables (t), base types (b), pure function types ($\sigma \to \tau$), or effectful function types ($\sigma/\alpha \to \tau/\beta$), which represent function types $\sigma \to \tau$ where the answer type changes from α to β.

Figure 3 defines the type system of λ^{ATM}. Type judgments are either $\Gamma \vdash_p e : \tau$ (pure judgments) or $\Gamma \vdash e : \tau; \alpha, \beta$ (effectful judgments), the latter of which means that evaluating e with the answer type α yields a value of type τ with the answer type being modified to β. The typing rules are based on Danvy

$$\tau, \sigma, \alpha, \beta ::= t \mid b \mid \sigma \to \tau \mid (\sigma/\alpha \to \tau/\beta)$$
$$A ::= \tau \mid \forall t.A$$
$$\Gamma ::= \emptyset \mid \Gamma, x : A$$

Figure 2: Types, Type Schemes and Type Environments

$$\frac{x : A \in \Gamma, \tau < A}{\Gamma \vdash_p \ x : \tau}\ \text{var} \qquad \frac{\Gamma \vdash e : \sigma; \sigma, \tau}{\Gamma \vdash_p \langle e \rangle : \tau}\ \text{reset}$$

$$\frac{\Gamma, x : \tau \to \alpha \vdash_p e : \beta}{\Gamma \vdash \mathcal{S}x.e : \tau; \alpha, \beta}\ \text{shift} \qquad \frac{(c \text{ is a constant of type } b)}{\Gamma \vdash_p c : b}\ \text{const}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau; \beta, \gamma}{\Gamma \vdash_p \lambda x.e : \sigma/\beta \to \tau/\gamma}\ \text{fun} \qquad \frac{\Gamma \vdash_p e : \tau}{\Gamma \vdash e : \tau; \alpha, \alpha}\ \text{exp}$$

$$\frac{\Gamma \vdash e_1 : \sigma/\alpha \to \tau/\beta; \beta, \gamma \qquad \Gamma \vdash e_2 : \sigma; \gamma, \delta}{\Gamma \vdash e_1 e_2 : \tau; \alpha, \delta}\ \text{app}$$

$$\frac{\Gamma \vdash_p e_1 : \sigma \to \tau \qquad \Gamma \vdash_p e_2 : \sigma}{\Gamma \vdash_p e_1 e_2 : \tau}\ \text{app-p}$$

$$\frac{\Gamma \vdash_p v : \sigma \qquad \Gamma, x : \mathrm{Gen}\,(\sigma; \Gamma) \vdash e : \tau; \alpha, \beta}{\Gamma \vdash \text{let } x = v \text{ in } e : \tau; \alpha, \beta}\ \text{let}$$

Figure 3: Typing Rules of the Source Calculus

and Filinski's [6] except that we have let-polymorphism and clear distinction of pure judgments from impure judgments following Asai and Kameyama [1].

In the var rule, $\tau < A$ means that the type τ is an instance of type scheme A, and the type $\mathrm{Gen}\,(\sigma; \Gamma)$ denotes $\forall t_1, \cdots \forall t_n.\sigma$ where $t_1, \cdots, t_n$ are the type variables that appear in σ but not appear in Γ freely. The delimited continuations captured by shift expressions are pure functions (they are polymorphic in answer types), and we use the pure function space $\tau \to \alpha$ for this purpose. On the contrary, the functions introduced by lambda are, in general, effectful. Accordingly, we have two rules for applications. Note that the body of a shift expression is restricted to a pure expression, in order to simplify the definition of our translation. This choice does not change the expressive power of the language, since we can always insert a reset into the topmost position of the body of a shift expression, to turn the body to a pure expression, without affecting typability and operational behavior. The exp rule turns pure terms into effectful terms.

The type system of the source calculus λ^{ATM} enjoys the subject reduction property. The proof is standard and omitted.

We then define the target calculus λ^{mpsr}, which is a polymorphic calculus with multi-prompt shift and reset (but without ATM). The calculus is similar, in spirit, to Gunter et al.'s calculus with the `cupto` and `set` operators [9]. Besides disallowing ATM, the target calculus differs from the source calculus in that the control operators are named, to allow mixing multiple effects in a single program. The names for control operators are called *prompts* for historical reasons, and denoted by $p, q, \cdots$. In our formulation, prompts are first-class values and can be bound to ordinary variables x. Prompts are given as prompt-constants, or can be generated dynamically by the $\mathcal{P}$ primitive. For instance, evaluating $\mathcal{P}x.\langle 1 + \mathcal{S}_x k.e\rangle_x$ first creates a fresh prompt p and substitutes it for x, then evaluate $\langle 1 + \mathcal{S}_p k.e\rangle_p$. This choice of the formulation closely follows Kiselyov's DelimCC library for multi-prompt shift/reset.

Types and typing environments are defined as follows:

$$\begin{aligned} \tau, \sigma &::= t \mid b \mid \sigma \to \tau \mid \tau \text{ pr} \\ A &::= \tau \mid \forall t.A \\ \Gamma &::= \emptyset \mid \Gamma, x : A \end{aligned}$$

where τ pr is the type for the prompts with the answer type τ. The syntax of values and terms are defines as follows:

$$\begin{aligned} v &::= x \mid c \mid \lambda x.e \mid p \\ e &::= v \mid e_1 e_2 \mid \mathcal{S}_v x.e \mid \langle e\rangle_v \mid \mathcal{P}x.e \mid \text{let } x = v \text{ in } e \mid \Omega \end{aligned}$$

where p is a prompt-constant. The control operators now receive not only prompt-constants, but values which will reduce to prompts. The term $\mathcal{P}x.e$ creates a fresh prompt and binds x to it. The term Ω denotes a non-terminating computation of arbitrary types. It may be defined in terms of shift, but for the sake of clarity, we added it as a primitive.

Evaluation contexts and evaluation rules are given as follows:

$$\begin{aligned} E &::= [\,] \mid Ee \mid vE \mid \langle E\rangle_p \\ E[(\lambda x.e)v] &\rightsquigarrow E[e\{v/x\}] \\ E[\text{let } x = v \text{ in } e] &\rightsquigarrow E[e\{v/x\}] \\ E[\mathcal{P}x.e] &\rightsquigarrow E[e\{p/x\}] \qquad p \text{ is a fresh prompt-constant} \\ E[\langle v\rangle_p] &\rightsquigarrow E[v] \\ E[\langle E_p[\mathcal{S}_p x.e]\rangle_p] &\rightsquigarrow E[\langle e\{\lambda y.\langle E_p[y]\rangle_p/x\}\rangle_p] \\ E[\Omega] &\rightsquigarrow E[\Omega] \end{aligned}$$

Note that we use E_p in the second last rule, which is an evaluation context that does not have a reset with the prompt p around the hole, and thus implies that we capture the continuation up to the nearest reset with the prompt p.

Finally we give typing rules for the target calculus in Figure 4. The type system of the target calculus is mostly standard except the use of prompts. In the shift rule, the prompt expression v must be of type σ pr where σ is the

$$\frac{x : A \in \Gamma, \tau < A}{\Gamma \vdash x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of } b)}{\Gamma \vdash c : b}\ \text{const}$$

$$\frac{\Gamma \vdash v : \tau\ \text{pr} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle_v : \tau}\ \text{reset} \qquad \frac{\Gamma \vdash v : \sigma\ \text{pr} \qquad \Gamma, x : \tau \to \sigma \vdash e : \sigma}{\Gamma \vdash \mathcal{S}_v\ x.e : \tau}\ \text{shift}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ \text{fun} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}\ \text{app}$$

$$\frac{\Gamma \vdash v : \sigma \qquad \Gamma, x : \text{Gen}\,(\sigma; \Gamma) \vdash e : \tau}{\Gamma \vdash \text{let } x = v \text{ in } e : \tau}\ \text{let} \qquad \frac{\Gamma, x : \sigma\ \text{pr} \vdash e : \tau}{\Gamma \vdash \mathcal{P}x.e : \tau}\ \text{prompt}$$

$$\frac{}{\Gamma \vdash \Omega : \tau}\ \text{omega}$$

Figure 4: Typing Rules of the Target Calculus

$$\begin{aligned}
[\![\tau; \alpha, \beta]\!] &= [\![\beta]\!]\ \text{pr} \to [\![\alpha]\!]\ \text{pr} \to [\![\tau]\!] \\
[\![b]\!] &= b \\
[\![t]\!] &= t \\
[\![\sigma \to \tau]\!] &= [\![\sigma]\!] \to [\![\tau]\!] \\
[\![\sigma/\alpha \to \tau/\beta]\!] &= [\![\sigma]\!] \to [\![\tau; \alpha, \beta]\!] \\
[\![\tau]\!] &= [\![\tau]\!] \\
[\![\forall t.A]\!] &= \forall t.[\![A]\!] \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}$$

Figure 5: Translation for Types, Type Schemes and Type Environments

type of the body of the shift expression. A similar restriction is applied to the reset rule. In the prompt rule, we can create an arbitrary prompt and binds a variable x to it.

The type system enjoys the subject reduction property modulo the set of dynamically created prompts which have infinite extents.

5 Translation

In this section, we give the syntax-directed translation from λ^{ATM} to λ^{mpsr}, which translates away the feature of answer-type modification. The translation borrows the idea of Kiselyov's implementation of typed printf in terms of multi-prompt shift and reset, but this paper gives a translation for the whole source calculus and also a proof of the type preservation property. Later, we will show a tagless-final implementation of our translation which is another evidence that our translation actually works type-safely.

Figure 6 presents the translation rules from λ^{ATM} to λ^{mpsr}. As we have explained in earlier sections, we emulate ATM from the type α to the type β in

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|c|\!) &= c \\
(\!|\lambda x.e|\!) &= \lambda x.[\![e]\!] \\
(\!|e_1 e_2|\!) &= (\!|e_1|\!)(\!|e_2|\!) \\
(\!|\langle e \rangle|\!) &= \mathcal{P}pq.\langle(\lambda y.\mathcal{S}_{q\_}.y)([\![e]\!]pq)\rangle_p \\
[\![e_1 e_2]\!] &= \lambda pq.\mathcal{P}rs.([\![e_1]\!]rs)([\![e_2]\!]pr)sq \\
[\![\text{let } x = v \text{ in } e_2]\!] &= \lambda pq.\text{let } x = (\!|v|\!) \text{ in } [\![e_2]\!]pq \\
[\![\mathcal{S}k.e]\!] &= \lambda pq.\mathcal{S}_p k'.((\lambda k.(\!|e|\!))(\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q) \\
[\![e]\!] &= \lambda pq.\mathcal{S}_p k.\langle k(\!|e|\!)\rangle_q \qquad e \text{ is a pure term}
\end{aligned}
$$

Figure 6: Translation for Terms

terms of two prompts whose answer types are α pr and β pr. Hence the triple $\tau; \alpha, \beta$ in the typing judgment is translated to the type $[\![\beta]\!]$ pr $\to [\![\alpha]\!]$ pr $\to [\![\tau]\!]$.

Figure 5 presents the translation rules for types, type schemes and type environments. They are translated in a natural way except that the type for effectful functions $\sigma/\alpha \to \tau/\beta$, which are translated to standard function types but their codomains are the translation of the triples above.

We can show that our translation preserves typing.

Theorem 1 (Type preservation)**.** *If $\Gamma \vdash e : \tau; \alpha, \beta$ is derivable in the source calculus λ^{ATM}, then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau; \alpha, \beta]\!]$ is derivable in the target calculus λ^{mpsr}.*

Similarly, if $\Gamma \vdash_p e : \tau$ is derivable in λ^{ATM}, so is $[\![\Gamma]\!] \vdash (\!|e|\!) : [\![\tau]\!]$ in λ^{mpsr}.

Proof. We will prove the two statements by simultaneous induction on the derivations. Here we only show a few interesting cases.

(Case $e = \langle e_1 \rangle$) We have a derivation for:

$$\frac{\Gamma \vdash e_1 : \sigma; \sigma, \tau}{\Gamma \vdash_p \langle e_1 \rangle : \tau}$$

By induction hypothesis, we can derive $[\![\Gamma]\!] \vdash [\![e_1]\!] : [\![\sigma; \sigma, \tau]\!]$. Let $\Gamma' = [\![\Gamma]\!], p : [\![\tau]\!]$ pr, $q : [\![\sigma]\!]$ pr and $\Gamma'' = \Gamma', y : [\![\sigma]\!]$. We have the following derivation:

$$\dfrac{\Gamma' \vdash p : [\![\tau]\!] \text{ pr} \qquad \dfrac{\dfrac{\dfrac{\Gamma'' \vdash q : [\![\sigma]\!] \text{ pr} \qquad \Gamma'' \vdash y : [\![\sigma]\!]}{\Gamma'' \vdash \mathcal{S}_{q\_}.y : [\![\tau]\!]}}{\Gamma' \vdash \lambda v.\mathcal{S}_{q\_}.y : [\![\sigma]\!] \to [\![\tau]\!]} \qquad \dfrac{\Gamma' \vdash q : [\![\sigma]\!] \text{ pr} \qquad \dfrac{\Gamma' \vdash p : [\![\tau]\!] \text{ pr} \qquad \Gamma' \vdash [\![e_1]\!] : [\![\sigma; \sigma, \tau]\!]}{\Gamma' \vdash [\![e_1]\!]p : [\![\sigma]\!] \text{ pr} \to [\![\sigma]\!]}}{\Gamma' \vdash [\![e_1]\!]pq : [\![\sigma]\!]}}{\Gamma' \vdash (\lambda y.\mathcal{S}_{q\_}.y)\,([\![e_1]\!]pq) : [\![\tau]\!]}}{\dfrac{\Gamma' \vdash \langle(\lambda y.\mathcal{S}_{q\_}.y)\,([\![e_1]\!]pq)\rangle_p : [\![\tau]\!]}{[\![\Gamma]\!] \vdash \mathcal{P}p.\mathcal{P}q.\langle(\lambda v.\mathcal{S}_{q\_}.v)\,([\![e_1]\!]pq)\rangle_p : [\![\tau]\!]}}$$

which derives $[\![\Gamma]\!] \vdash (\!|\langle e_1 \rangle|\!) : [\![\tau]\!]$.

(Case $e = \mathcal{S}x.e_1$) We have a deviation for

$$\frac{\Gamma, x : \tau \to \alpha \vdash_p e_1 : \beta}{\Gamma \vdash \mathcal{S}x.e_1 : \tau; \alpha, \beta}$$

By induction hypothesis $[\![\Gamma, x : \tau \to \alpha]\!] \vdash (\!|e_1|\!) : [\![\beta]\!]$ is derivable. Let $\Gamma' = [\![\Gamma]\!], p : [\![\beta]\!]$ pr, $q : [\![\alpha]\!]$ pr, $\Gamma'' = \Gamma', k' : [\![\tau]\!] \to [\![\beta]\!]$, and $\Gamma''' = \Gamma'', y : [\![\tau]\!]$, then we have:

$$\dfrac{\Gamma'' \vdash p : [\![\beta]\!] \text{ pr} \qquad \dfrac{\dfrac{\dfrac{\Gamma'', x : [\![\tau]\!] \to [\![\alpha]\!] \vdash (\!|e_1|\!) : [\![\beta]\!]}{\Gamma'' \vdash \lambda x.(\!|e_1|\!) : ([\![\tau]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]} \qquad \dfrac{\Gamma''' \vdash \langle (\lambda\_.\Omega)\,(k'y)\rangle_q : [\![\alpha]\!]}{\Gamma'' \vdash \lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q : [\![\tau]\!] \to [\![\alpha]\!]}}{\Gamma'' \vdash (\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\beta]\!]}}{\Gamma' \vdash \mathcal{S}_p k'.\,(\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\tau]\!]}}{[\![\Gamma]\!] \vdash \lambda p.\lambda q.\mathcal{S}_p k'.\,(\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\beta]\!] \text{ pr} \to [\![\alpha]\!] \text{ pr} \to [\![\tau]\!]}$$

which derives $[\![\Gamma]\!] \vdash [\![\mathcal{S}x.e_1]\!] : [\![\tau; \alpha, \beta]\!]$. (end of proof)

Hence our translation preserves typing. We think that our translation also preserve operational semantics but its formalization is left for future work.

6 Tagless-final embedding in OCaml

We have implemented interpreters for the calculus in Figure 3 via the translation presented in the previous section. To simplify the implementation, we have eliminated let polymorphism from the source calculus.

Our implementation is based on Carette et al.'s tagless-final approach [5], which lets one embed a domain-specific language in a metalanguage while preserving typing of the embedded language. The syntax as well as the typing rules of the embedded language are represented by a signature, and its semantics is given as an interpretation of this signature. In this approach, type checking/inference of the embedded language is reduced to that of the metalanguage, and all representable terms and interpretations are type safe, by construction.

In this work, we choose OCaml plus multi-prompt shift and reset (implemented in the DelimCC library) as the metalanguage. We embed our source calculus λ^{ATM} in this metalanguage, and give its semantics using our translation.

Figure 7 shows the module signature (called Symantics) for our source calculus λ^{ATM}, embedded in OCaml. Each function in the signature represents a term-constructor of λ^{ATM}, and its type encodes the corresponding typing rule. The type `'t pure` means that the expression is pure and has the type `'t`. The type `('t, 'a, 'b) eff` means that the expression has the type `'t` and has an effect that modifies the answer type from `'a` to `'b`.

Note that, we can represent all and only typable terms in λ^{ATM} as a combination of the functions defined in the signature, and the typability of embedded terms are checked by the type system of OCaml; in other words, all representable terms are typable *when they are constructed*.

In the tagless-final approach, operational semantics of the embedded language is given as an interpretation of the Symantics signature, namely, a module

```
module type Symantics = sig
  type 't pure
  type ('t, 'a, 'b) eff
  type ('s, 't, 'a, 'b) efun
  type ('s, 't) pfun

  val const : 't -> 't pure
  val lam : ('s pure -> ('t, 'a, 'b) eff)
       -> ('s, 't, 'a, 'b) efun pure
  val app : (('s, 't, 'a, 'b) efun, 'b, 'c) eff
       -> ('s, 'c, 'd) eff -> ('t, 'a, 'd) eff
  val appP : ('s, 't) pfun pure -> 's pure -> 't pure
  val shift : (('t, 'a) pfun pure -> 'b pure)
       -> ('t, 'a, 'b) eff
  val reset : ('s, 's, 't) eff -> 't pure
  val exp : 't pure -> ('t, 'a, 'a) eff
  val run : 't pure -> 't
end
```

Figure 7: Signature of the Embedded Language

of type Symantics. An interpreter for our source language is shown in Figure 10 in the appendix. The interpreter interprets an embedded term in the meta-language, using the translation introduced in this paper.

To write concrete examples, we extend the source language with primitive functions, conditionals, and the fixpoint operator. Its signature SymP is shown in Figure 9 in the appendix of this paper.

Using these modules, we can write programming examples such as list append and prefix, shown in Figure 8 where we use syntax sugars for list-manipulating functions defined in Figure 12. Running these examples gives correct answers, as shown in the comments of the code in Figure 8.

In summary, we have successfully implemented our source language and the translation into the target language using the tagless-final approach. We think that this implementation provides another good evidence that our translation is type preserving. Thanks to the tagless-final approach, our implementation is extensible, and in fact, we added several primitive such as the fixpoint operator in a type-safe way.

7 Related Work and Conclusion

In this paper, we have proposed type-preserving translation for embedding programs with ATM into those without. Our translation uses multi-prompt systems and dynamic creation of prompts to emulate two answer types in effectful terms. We proved type preservation and implemented in OCaml the translation using the tagless-final style, which would add the "credibility" of our work.

```
module Example (S: SymP) = struct
 open LIST  (* LIST module defines syntax sugars for lists.
    *)

 let append = fixE (fun f x ->
     ifE (null x)
       (shift (fun k -> k))
       (head (exp x) @* app (exp f) (tail @@ exp x)))
 let res1 = run @@
   appP (reset (app (exp append) (exp @@ list [1;2;3])))
        (list [4;5;6])
 (* res1 returns [1; 2; 3; 4; 5; 6] *)

 let prefix =
   fixE (fun f x ->
     ifE (null x)
       (shift (fun k -> list []))
       (head (exp x) @* shift (fun k -> reset @@
              (exp (appP k (list [])))
               @* (exp (reset @@ app (exp2 k)
                         (app (exp f) (tail @@ exp x)))))))

 let res2 = run @@
   reset @@ app (exp prefix) (exp @@ list [1;2;3])
 (* res2 returns [[1]; [1; 2]; [1; 2; 3]] *)
end
```

Figure 8: Programming Examples

Let us briefly summarize related work. Rompf et al. [12] implemented shift and reset in Scala, that allow answer-type modification. Their source language needs relatively heavy type annotations to be implemented by a selective CPS transformation, and does not allow higher-order functions. Masuko and Asai [11] designed a language OchaCaml, which is Caml light extended with shift and reset. OchaCaml fully supports ATM at the cost of redesigning the whole type system and an extension of the run-time system. Wadler [13] studied monad-like structures to express shift and reset with Danvy and Filinski's type system [6]. Inspired by his work, Atkey [2, 3] proposed *parameterised monads*, which generalize monads. Parametrised monads take two more type parameters to express inputs and outputs, or initial states and final states. Unlike standard monads, parameterised monads can express answer-type modification. He studied categorical foundation of parameterised monads. Kiselyov [10] independently studied the same notion as parameterised monads, and gave an implementation and a number of programming examples using it.

For future work, we plan to formally prove the semantics-preservation property mentioned in this paper. Investigating other delimited-control operators

such as shift0/reset0 and control/prompt with answer-type modification would be also interesting.

Acknowledgments: We are grateful to the anonymous reviewers for their constructive comments. The first and second authors are supported in part by JSPS Grant-in-Aid for Scientific Research B (No. 25280020).

References

[1] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proc. Fifth Asian Symposium on Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin Heidelberg, 2007. 2, 5, 6

[2] Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, Kuressare, Estonia, 2006. eWiC. 12

[3] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, June 2009. 12

[4] Malgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, page 289, 2009. 5

[5] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509, April 2009. 10

[6] Olivier Danvy and Andrzej Filinski. A Functional Abstraction of Typed Contexts. Technical report, Computer Science Department, University of Copenhagen, 1989. 5, 6, 12

[7] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, USA, 1990. ACM Press. 1

[8] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, New York, USA, 1988. ACM Press. 1

[9] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, New York, USA, 1995. ACM Press. 7

[10] Oleg Kiselyov. Parametrized 'monad'. `http://okmij.org/ftp/Computation/monads.html#param-monad`, 2006. 12

[11] Moe Masuko and Kenichi Asai. Caml Light + shift/reset = Caml Shift. In *First International Workshop on the Theory and Practice of Delimited Continuations*, pages 33–46, 2011. 1, 12

[12] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, page 317, New York, USA, 2009. ACM Press. 1, 12

[13] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994. 12

A Implementation of Interpreters

We show our tagless-final implementation of the source language and its interpretation, which is based on our translation in the paper.

Figure 9 shows the signature for the extended language.

```
module type SymP = sig
 include Symantics

 val p2E : ('t1 -> 't2 -> 't3) -> ('t1, 'a, 'b) eff
     -> ('t2, 'b, 'c) eff -> ('t3, 'a, 'c) eff
 val pE: ('t -> 'u) -> ('t, 'a, 'a) eff
     -> ('u, 'a, 'a) eff
 val ifE : (bool, 'b, 'c) eff -> ('t, 'a, 'b) eff
     -> ('t, 'a, 'b) eff -> ('t, 'a, 'c) eff
 val fixE : ((('s, 't, 'a, 'b) efun) pure -> 's pure
     -> ('t, 'a, 'b) eff) -> ('s, 't, 'a, 'b) efun pure
end
```

Figure 9: Signature of the Extended Language

Figure 10 is an interpreter for the embedded language, which is a Symantics module. As can be seen, our implementation reflects our translation to the target language, which is OCaml plus DelimCC.

Figure 11 shows an interpreter for the extended language, which is a SymP module.

We use syntax sugar for list-manipulating functions, which are defined in the LIST module in Figure 12.

Here pE and p2E are primitives for lifting up one or two arguments primitives, and fixE, ifE and exp2 are the fixpoint operator, conditional, and a primitive to turn pure functions to effectful one, respectively.

```
module R = struct
  type 't pure = unit -> 't
  type ('t, 'a, 'b) eff = 'b prompt -> 'a prompt -> 't
  type ('s, 't, 'a, 'b) efun = 's pure -> ('t, 'a, 'b)
   eff
  type ('s, 't) pfun = 's pure -> 't pure

  let coerce _ = failwith "unreachable"

  let wrap x : 'a pure = fun () -> x
  let unwrap : 'a pure -> 'a = fun x -> x ()

  let const x = wrap x
  let lam f = wrap f

  let app e1 (e2: ('s, 'c, 'd ) eff) : ('t, 'a, 'd) eff =
    fun p q ->
    let r, s = new_prompt (), new_prompt () in
    let v2 = wrap @@ e2 p r in
    let v1 = e1 r s in
    v1 v2 s q

  let appP e1 e2 : 't pure =
    fun () ->
    let v2 = unwrap e2 in
    let v1 = unwrap e1 in
    v1 (wrap v2) ()

  let shift f : ('t, 'a, 'b) eff =
    fun p q ->
    Delimcc.shift p (fun k' ->
      unwrap @@ f @@ wrap (fun y ->
        wrap @@ push_prompt q
                  (fun () -> coerce (k' @@ unwrap y ))))

  let reset e =
    fun () ->
      let p, q = new_prompt (), new_prompt () in
      push_prompt p (fun () -> abort q @@ e p q)

  let exp e =
    fun p q -> Delimcc.shift p
      (fun k -> push_prompt q (fun () -> k @@ unwrap e))

   let run (x: 't pure) : 't = x ()
 end
```

Figure 10: Interpretation of the Embedded Language

```
module RP = struct
 include R

 let p2E f e1 e2 = fun p q ->
   let r = new_prompt () in
   let v2 = e2 p r in
   let v1 = e1 r q in
   f v1 v2

 let pE f x = fun p q -> f (x p q)

 let ifE b e e' = fun p q ->
   let r = new_prompt () in
   if b p r then e r q else e' r q

 let rec fixE f : ('s, 't, 'a, 'b) efun pure =
   lam (fun x -> f (fun x -> fixE f x) x)
end
```

Figure 11: Interpretation of the Extended Language

```
module LIST (S: SymP) = struct
 open S

 let list x = const x
 let (@* ) x y = p2E (fun x y -> x :: y) x y
 let head x = pE List.hd x
 let tail x = pE List.tl x
 let null x = pE (fun x -> x = []) @@ exp x
 let exp2 f = exp @@ lam (fun x -> exp @@ appP f x)
end
```

Figure 12: Syntax sugar for Lists

Logical by need

Pierre-Marie Pédrot & Alexis Saurin
Université Paris Diderot – Paris 7
France

12 April 2015

Call-by-need calculi are complex to design and reason with. When adding control effects, the very notion of canonicity is irremediably lost, the resulting calculi being necessarily ad hoc. This calls for a design of call-by-need guided by logical rather than operational considerations. This would allow for a direct extension to control operators, given their strong connections with classical logic. This work provides such logical by-need calculi rooted in linear head reduction.

After recalling linear head reduction, it is first reformulated thanks to closure contexts stemming from Danos and Regnier's sigma-equivalence. This reformulation allows to extend linear head reduction to the lambda-mu-calculus.

From the linear head reduction, a call-by-need calculus is then derived in three main steps. This methodology is eventually validated by the design of a classical by-need calculus, that is a lazy lambda-mu-calculus.