A Modular Structural Operational Semantics for Delimited Continuations

Neil Sculthorpe
PLANCOMPS Project
Department of Computer Science
Swansea University, UK
N.A.Sculthorpe@swansea.ac.uk

Paolo Torrini
GRACEFUL Project
Department of Computer Science
KU Leuven, Belgium
Paolo.Torrini@cs.kuleuven.be

Peter D. Mosses
PLANCOMPS Project
Department of Computer Science
Swansea University, UK
P.D.Mosses@swansea.ac.uk

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*.

1 Introduction

Modular Structural Operational Semantics (MSOS) [25, 26, 27] is a variant of the well-known Structural Operational Semantics (SOS) framework [29]. The principal innovation of MSOS relative to SOS is that it allows the semantics of a programming construct to be specified independently of any semantic 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, 25], 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. [30, 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* [18, 19, 34] and *shift* [12, 13, 14], 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. [18, 22, 24, 33]), ours is 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. We will then present our MSOS specification of the dynamic semantics of delimited control operators (Section 4). To ensure that our MSOS specification does indeed define the same control operators as described in the literature, we provide a proof of equivalence between our specification and one based on evaluation contexts (Section 5).

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].

This is the corrected version (9th September 2016) of an article published in EPTCS.

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) [18, 19, 34] and *shift* (associated with the *reset* delimiter) [12, 13, 14], 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 the expression evaluates to 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*:

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

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 [32], and in the literature [18, 19].

3 Modular SOS

The rules in this paper will be presented using *Implicitly Modular SOS* (I-MSOS) [27], 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, 29]) and the basics of MSOS [25, 26, 27].

The key notational convenience of I-MSOS is that any semantic 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 semantic 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 the conclusion. Observable entities are required to have a default value, which is implicitly used in the conclusion of rules that lack a premise and do not mention the entity. Note that by *premise*, we refer specifically to a transition of the $\rightarrow$ relation, not any side conditions on the rule (which, for notational convenience, we also write above the line).

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. Funcons facilitate formal specification of programming languages by serving as a target language for a specification given 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, 28].

We will now present some examples of funcons, and their specifications as small-step I-MSOS rules. No familiarity with the funcon framework is required: for the purposes of understanding this paper the funcons may simply be regarded as abstract syntax. We typeset funcon names in **bold**, meta-variables in *Capitalised Italic*, and the names of semantic entities in sans-serif. When we come to funcons for control operators, we will continue to use *italic* when referring to the control operator in general, and **bold** when referring to the funcon specifically.

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 absence of an exception, or $\mathbf{some}(V)$, denoting the occurrence of an exception with value V. The side condition $val(V)$ 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.

$$\frac{E \rightarrow E'}{\mathbf{throw}(E) \rightarrow \mathbf{throw}(E')} \quad (1)$$

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

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

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

$$\frac{val(V)}{\mathbf{catch}(V,H) \rightarrow V} \quad (5)$$

Figure 1: I-MSOS rules for exception handling.

$$\frac{\rho(I) = V}{\mathsf{env}\ \rho \vdash \mathbf{bv}(I) \to V} \quad (6)$$

$$\mathsf{env}\ \rho \vdash \mathbf{lambda}(I,E) \to \mathbf{closure}(\rho,I,E) \quad (7)$$

$$val(\mathbf{closure}(\rho,I,E)) \quad (8)$$

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

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

$$\frac{val(V_1) \qquad val(V_2)}{\mathbf{apply}(\mathbf{closure}(\rho,I,V_1),V_2) \to V_1} \quad (11)$$

$$\frac{val(V) \qquad \mathsf{env}\ (\{I \mapsto V\}/\rho) \vdash E \to E'}{\mathsf{env}\ \_ \vdash \mathbf{apply}(\mathbf{closure}(\rho,I,E),V) \to \mathbf{apply}(\mathbf{closure}(\rho,I,E'),V)} \quad (12)$$

Figure 2: I-MSOS rules for call-by-value lambda calculus.

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 8), 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.

Again, observe that rules 9–11 do not mention the environment env; it is propagated implicitly. Furthermore, consider that none of the rules in Figure 1 mention the environment env, and none of the rules in Figure 2 mention the exc signal. However, the modular nature of I-MSOS specifications allows the two sets of rules to be combined without modification, with implicit propagation handling the unmentioned entities. For comparison, in Figure 3 we present a conventional SOS specification of this call-by-value lambda calculus combined with exception handling, in which both semantic entities are mentioned explicitly in every rule.

$$\frac{\mathsf{env}\ \rho \vdash E \xrightarrow{\mathsf{exc}\ X} E'}{\mathsf{env}\ \rho \vdash \mathbf{throw}(E) \xrightarrow{\mathsf{exc}\ X} \mathbf{throw}(E')}$$

$$\frac{val(V)}{\mathsf{env}\ \rho \vdash \mathbf{throw}(V) \xrightarrow{\mathsf{exc}\ \mathbf{some}(V)} \mathbf{stuck}}$$

$$\frac{\mathsf{env}\ \rho \vdash E \xrightarrow{\mathsf{exc}\ \mathbf{none}} E'}{\mathsf{env}\ \rho \vdash \mathbf{catch}(E,H) \xrightarrow{\mathsf{exc}\ \mathbf{none}} \mathbf{catch}(E',H)}$$

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

$$\frac{val(V)}{\mathsf{env}\ \rho \vdash \mathbf{catch}(V,H) \xrightarrow{\mathsf{exc}\ \mathbf{none}} V}$$

$$\frac{\rho(I) = V}{\mathsf{env}\ \rho \vdash \mathbf{bv}(I) \xrightarrow{\mathsf{exc}\ \mathbf{none}} V}$$

$$\mathsf{env}\ \rho \vdash \mathbf{lambda}(I,E) \xrightarrow{\mathsf{exc}\ \mathbf{none}} \mathbf{closure}(\rho,I,E)$$

$$\frac{\mathsf{env}\ \rho \vdash E_1 \xrightarrow{\mathsf{exc}\ X} E_1'}{\mathsf{env}\ \rho \vdash \mathbf{apply}(E_1,E_2) \xrightarrow{\mathsf{exc}\ X} \mathbf{apply}(E_1',E_2)}$$

$$\frac{val(V) \qquad \mathsf{env}\ \rho \vdash E \xrightarrow{\mathsf{exc}\ X} E'}{\mathsf{env}\ \rho \vdash \mathbf{apply}(V,E) \xrightarrow{\mathsf{exc}\ X} \mathbf{apply}(V,E')}$$

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

$$\frac{val(V_1) \qquad val(V_2)}{\mathsf{env}\ \rho \vdash \mathbf{apply}(\mathbf{closure}(\rho,I,V_1),V_2) \xrightarrow{\mathsf{exc}\ \mathbf{none}} V_1}$$

Figure 3: SOS rules for lambda calculus with exception handling.

4 I-MSOS Specifications of Control Operators

We now present a dynamic semantics for control operators in the MSOS framework. We 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

Whether the semantics of control operators can be specified using MSOS has been considered an open problem ([30, Section 2]). We suspect that this is because there is no explicit representation of a term's context in the MSOS framework—any given rule only has access to the current subterm and the contents of any semantic entities—so it is not immediately obvious how to capture the context as an abstraction.

Our approach is to construct the current continuation of a control operator in the rule for its enclosing delimiter. We achieve this by exploiting the way that a small-step semantics, for each step of computation, builds a derivation tree from the root of the program term 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 part of the derivation, but so too will a rule for the enclosing delimiter. At each such step, the current continuation corresponds to an abstraction of the control operator (and its argument) from the subterm of the enclosing delimiter, and thus can be constructed from that subterm.

We represent reified continuations as first-class abstractions, using the **lambda** funcon from Section 3. Constructing the abstraction is achieved in two stages: the rule for **control** replaces the occurrence of **control** (and its argument) with a fresh identifier, and the rule for **prompt** constructs the abstraction from the updated subterm. At a first approximation, this suggests the following rules:

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

$$\frac{E \xrightarrow{\mathsf{ctrl}\ \mathbf{some}(F,I)} E' \qquad K = \mathbf{lambda}(I, E')}{\mathbf{prompt}(E) \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} \mathbf{prompt}(\mathbf{apply}(F, K))} \tag{14}$$

The side condition *fresh-id*(I) requires that the identifier I introduced by this rule does not already occur in the program. Rule 13 replaces the term $\mathbf{control}(F)$ with $\mathbf{bv}(I)$, and emits a signal (ctrl) 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 subterm E' (which will now contain $\mathbf{bv}(I)$ in place of $\mathbf{control}(F)$).

4.2 The Auxiliary Environment

There is one problem with the approach we have just outlined, which is that the identifier I is introduced dynamically when the control operator executes, by which time closures may have already formed. In particular, if **control** occurs inside the body of a **lambda**, and the enclosing **prompt** is outside that **lambda**, then the $\mathbf{bv}(I)$ funcon would be introduced inside a **closure** that has already formed, and hence

does not contain a binding for I. For example, consider the evaluation of the following term:

$$\begin{array}{ll} & \mathbf{prompt}(\mathbf{apply}(\mathbf{lambda}(x,\mathbf{control}(\mathbf{lambda}(k,\mathbf{apply}(\mathbf{bv}(k),2)))),1)) \\ \rightarrow & \quad \{\, by\ (7)\ \} \\ & \mathbf{prompt}(\mathbf{apply}(\mathbf{closure}(\emptyset,x,\mathbf{control}(\mathbf{lambda}(k,\mathbf{apply}(\mathbf{bv}(k),2)))),1)) \\ \rightarrow & \quad \{\, by\ (14)\ \} \\ & \mathbf{prompt}(\mathbf{apply}(\mathbf{lambda}(k,\mathbf{apply}(\mathbf{bv}(k),2)),\mathbf{lambda}(i,\mathbf{apply}(\mathbf{closure}(\emptyset,x,\mathbf{bv}(i)),1)))) \end{array}$$

The occurrence of $\mathbf{bv}(i)$ is now inside a closure containing an empty environment. After several more steps, evaluation of this term would get stuck when attempting to evaluate the body of that closure, as Rule 12 would provide an environment containing only x, which Rule 6 could not match.

This problem arises as a consequence of our choice to specify the semantics of lambda calculus using environments and closures. If we had instead given a semantics using substitution, then this problem would not have arisen. However, we prefer to use environments because they enable a more modular specification: a substitution-based semantics requires substitution to be defined over every construct in the language. Moreover, environments allow straightforward semantics for dynamic scope.

Our solution is to introduce an auxiliary environment that is not captured in closures. Figure 4 specifies $\mathbf{aux\text{-}bv}(I)$, which looks up the identifier I in this auxiliary environment, and $\mathbf{aux\text{-}let\text{-}in}(I,V,E)$, which binds the identifier I to the value V in the auxiliary environment and scopes that binding over the expression E. We make use of these funcons in the next subsection, where we give our complete specification of **control** and **prompt**.

$$\frac{\mu(I) = V}{\textsf{aux-env}\ \mu \vdash \mathbf{aux\text{-}bv}(I) \rightarrow V} \tag{15}$$

$$\frac{E_1 \rightarrow E_1'}{\mathbf{aux\text{-}let\text{-}in}(I,E_1,E_2) \rightarrow \mathbf{aux\text{-}let\text{-}in}(I,E_1',E_2)} \tag{16}$$

$$\frac{val(V) \qquad \textsf{aux-env}\ (\{I \mapsto V\}/\mu) \vdash E \rightarrow E'}{\textsf{aux-env}\ \mu \vdash \mathbf{aux\text{-}let\text{-}in}(I,V,E) \rightarrow \mathbf{aux\text{-}let\text{-}in}(I,V,E')} \tag{17}$$

$$\frac{val(V_1) \qquad val(V_2)}{\mathbf{aux\text{-}let\text{-}in}(I,V_1,V_2) \rightarrow V_2} \tag{18}$$

Figure 4: I-MSOS rules for bindings in the auxiliary environment.

4.3 Dynamic Semantics of *control* and *prompt*

We specify **control** as follows:

$$\frac{E \rightarrow E'}{\mathbf{control}(E) \rightarrow \mathbf{control}(E')} \tag{19}$$

$$\frac{val(F) \qquad fresh\text{-}id(I)}{\mathbf{control}(F) \xrightarrow{\textsf{ctrl}\ \mathbf{some}(F,I)} \mathbf{aux\text{-}bv}(I)} \tag{20}$$

Rule 19, in combination with the $val(F)$ premise on Rule 20, ensures that the argument function is evaluated to a closure before Rule 20 can be applied. Notice that Rule 20 uses **aux-bv**, in contrast to the preliminary Rule 13 which used **bv**.

We then specify **prompt** as follows:

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

$$\frac{E \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} E'}{\mathbf{prompt}(E) \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} \mathbf{prompt}(E')} \tag{22}$$

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

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

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 auxiliary-environment funcons from figures 2 and 4. These rules are modular: they are valid independently of whether the control operators coexist with a mutable store, exceptions, input/output signals, or other semantic entities. Except for the use of an auxiliary environment, our rules correspond closely to those in specifications of *control* and *prompt* based on evaluation contexts [18, 24]. However, our rules communicate between **control** and **prompt** by emitting signals, and thus do not require evaluation contexts. In Section 5, we present a proof of equivalence between our specification and a conventional one based on 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 analogous 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:

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

$$\frac{val(F) \qquad \textit{fresh-id}(K) \qquad \textit{fresh-id}(X)}{\mathbf{shift}(F) \to \mathbf{control}(\mathbf{lambda}(K, \mathbf{apply}(F, \mathbf{lambda}(X, \mathbf{reset}(\mathbf{apply}(\mathbf{bv}(K), \mathbf{bv}(X)))))))} \tag{25}$$

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**:

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

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

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

$$\frac{E \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} E'}{\mathbf{reset}(E) \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} \mathbf{reset}(E')} \tag{28}$$

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

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**:

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

This I-MSOS specification in Rules 27–30 is similar to the evaluation-context based specification of *shift* and *reset* in [24, 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 [34, 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:

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

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

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

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

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

4.6 Other Control Effects

In Section 3 we presented a direct specification of exception handling using a dedicated semantic 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 [34]), in which case the delimiters and semantic 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 [13]. These are beyond the scope of this paper, and remain an avenue for future work.

5 Adequacy

Our SOS model of call-by-value lambda calculus extended with delimited control, which we have presented using I-MSOS rules, is provably equivalent to one based on the reduction semantics (RS) of lambda terms where the evaluation strategy is specified using evaluation contexts. Reduction models for delimited control based on evaluation contexts were originally introduced in [18] and refined in [24]. The adequacy proof in this section (Prop. 8) is carried out with respect to our version of those models in a formalism that we call RC.

Our SOS model differs from reduction models in the framework it relies on. In particular, our SOS model uses environments and signals, whereas RC uses substitution and evaluation contexts. Moreover, there is a difference in the notion of value: in our SOS model function application is computed using closures, whereas RS uses β-reduction and substitution. In order to focus on the operational content of the models, it is convenient to get above these differences. We achieve this by embedding SOS in RS with explicit congruence rules (an embedding that we call LS), and by lifting RC to an environment-based formalism (called LR). We define a notion of adequacy between two systems, as an input-output relation, parametric in a translation. We show adequacy of two systems by proving that they are derivationally equivalent (in the sense of a step-wise relation), reasoning by induction on the structure of derivations. Our adequacy proof for SOS and RC is split into three main parts: the equivalence of SOS and LS, of LS and LR, and of LR and RC. A more challenging approach would involve giving a formal derivation of an RC model from SOS along the lines of [11], but that goes beyond the scope of this paper.

Here we intend to focus on equivalence with respect to delimited control. Given the equivalence between SOS and RC with respect to call-by-value lambda calculus (λV), we show that SOS and RC are equivalent with respect to the extension of λV with delimited control (λDC). More specifically, we define a syntactic representation of environments (standard and auxiliary ones) using contexts and lambda terms. We use this representation to define LS as a lambda-term encoding of SOS. Adequacy between SOS and LS is provable with respect to a simple translation relation.

We define LR as an environment-based version of RC obtained by lambda-lifting. We consider two distinct extensions of the LR model of λV with delimited control. The first one, which we call LR-DC, uses the lifted control rules of the original RC model, and thus equivalence with the RC model is straightforward. The second one, which we call LX-DC, uses the LS version of the SOS control rules. The difference between the LS model of λDC and LX-DC, which are provably equivalent, boils down to that between SOS transitions, based on congruence rules and also using closures, and RC transitions, based on evaluation contexts and using only lambda expressions. The adequacy of LX-DC and LR-DC, proved with respect to the identity translation (Prop. 7), gives us the result of primary interest.

5.1 Reduction Semantics

Our presentation of reduction semantics with evaluation contexts (RC) follows the main lines of the λDC model in [24]. Under the assumption that we only evaluate closed expressions, values and terms can be defined as follows:

$$V \;=\; \mathsf{lambda}(I,E) \tag{35}$$

$$E \;=\; V \mid \mathsf{bv}(I) \mid \mathsf{apply}(E,E) \mid \mathsf{control}(E) \mid \mathsf{prompt}(E) \tag{36}$$

A general notion of a context as a term with a hole can be defined by the following grammar:

$$C \;=\; [] \mid \mathsf{lambda}(I,C) \mid \mathsf{apply}(C,E) \mid \mathsf{apply}(E,C) \mid \mathsf{prompt}(C) \mid \mathsf{control}(C) \tag{37}$$

The call-by-value (CBV) evaluation strategy can be specified using the more restrictive notion of a CBV context (Q):

$$Q \;=\; [] \mid \mathsf{apply}(Q,E) \mid \mathsf{apply}(V,Q) \mid \mathsf{prompt}(Q) \mid \mathsf{control}(Q) \tag{38}$$

In order to represent delimited continuations, an even more restrictive notion is needed: a *pure* context (P-context), which is a CBV-context that does not include control delimiters [24]:

$$P \;=\; [] \mid \mathsf{apply}(P,E) \mid \mathsf{apply}(V,P) \mid \mathsf{control}(P) \tag{39}$$

The meta-linguistic notation $C[E]$ ($Q[E]$, $P[E]$) is used to represent a term factored into a context and the subterm that fills the hole—we can think of this as a form of term annotation. This factorisation is unique for the cases that we are considering.

The only reduction rules needed to specify λV are β-reduction and context propagation. These can be presented as follows (giving us the RC-V model), using $\{\_ \mapsto \_\}$ as meta-level notation for capture-avoiding uniform substitution:

$$\mathsf{apply}(\mathsf{lambda}(I,E),V) \longrightarrow E\{\mathsf{bv}(I) \mapsto V\} \tag{40}$$

$$\frac{E \longrightarrow E'}{Q[E] \longrightarrow Q[E']} \tag{41}$$

The reduction rules for prompt and control can be formulated as follows (giving us the RC-DC model), making use of P-contexts:

$$\mathsf{prompt}(V) \longrightarrow V \tag{42}$$

$$\frac{\mathit{val}(F) \qquad K = \mathsf{lambda}(I,P[\mathsf{bv}(I)]) \qquad \mathit{fresh\text{-}id}(I)}{\mathsf{prompt}(P[\mathsf{control}(F)]) \longrightarrow \mathsf{prompt}(\mathsf{apply}(F,K))} \tag{43}$$

In a system based on reduction semantics, *observational equivalence* can be defined as the smallest congruence relation $\equiv$ on terms that extends reduction equivalence with functional extensionality, i.e. such that

$$\frac{\forall\, V.\ \mathsf{apply}(F,V) \equiv \mathsf{apply}(F',V)}{F \equiv F'}$$

In presenting models based on RS, we typeset all construct names in sans-serif. We refer to SOS values as $\mathit{Val}_{\mathsf{SOS}}$ and to RC ones as $\mathit{Val}_{\mathsf{RC}}$. When needed, we subscript $\longrightarrow$ and $\equiv$ accordingly. We define *derivational equivalence* and *adequacy* with respect to a translation relation (not mentioned in the case that it is an identity), as follows.

Def. 1 Given two systems A and B, respectively defined on languages L_A and L_B with values Val_A and Val_B, and a one-to-one relation $R \subseteq (L_A, L_B)$, we say that

1. A and B are *adequate* with respect to R ($A \sim^R B$) whenever the following hold:
   A) If $E \longrightarrow_A^* V$, with $V \in Val_A$, then there exist $E'_a, E''_a \in L_A$, $E'_b, E''_b \in L_B$, $V' \in Val_B$ s.t. $E \equiv_A E'_a$, $V \equiv_A E''_a$, $R(E'_a, E'_b)$, $R(E''_a, E''_b)$, $E''_b \equiv_B V'$ and $E'_b \longrightarrow_B^* V'$
   B) If $E \longrightarrow_B^* V$, with $V \in Val_B$, then there exist $E'_a, E''_a \in L_A$, $E'_b, E''_b \in L_B$, $V' \in Val_A$ s.t. $E \equiv_B E'_b$, $V \equiv_B E''_b$, $R(E'_a, E'_b)$, $R(E''_a, E''_b)$, $E''_a \equiv_A V'$ and $E'_a \longrightarrow_B^* V'$
2. A and B are *derivationally equivalent* with respect to R whenever the following hold:
   A) $E_1 \longrightarrow_A E_2$ whenever there exist E_3, E_4, E'_1, E'_2, E'_3, E'_4 s.t. $E_1 \equiv_A E_3$, $E_2 \equiv_A E_4$, $R(E_3, E'_3)$, $R(E_4, E'_4)$, $E'_3 \equiv_B E'_1$, $E'_4 \equiv_B E'_2$ and $E'_1 \longrightarrow_B^* E'_2$
   B) $E_1 \longrightarrow_B E_2$ whenever there exist E_3, E_4, E'_1, E'_2, E'_3, E'_4 s.t. $E_1 \equiv_B E_3$, $E_2 \equiv_B E_4$, $R(E'_3, E_3)$, $R(E'_4, E_4)$, $E'_3 \equiv_A E'_1$, $E'_4 \equiv_A E'_2$ and $E'_1 \longrightarrow_A^* E'_2$

We define relational composition as $R_1 \circ R_2 = \lambda xy.\ \exists z.\ R_1(x,z) \wedge R_2(z,y)$.

5.2 Representing SOS as LS

In this section we define LS, as an encoding of SOS in lambda terms. Unlike reduction semantics, our SOS models rely internally on a linguistic extension to account for closures and the auxiliary environment notation. For this reason, we need an extended *internal* language, including the following additional constructs: $\mathsf{closure}(\rho, I, E)$ for closures, $\mathsf{aux\text{-}bv}(I)$ for auxiliary identifier lookup, and $\mathsf{aux\text{-}let\text{-}in}(I, E, E)$ for auxiliary let bindings; these constructs are not meant to be included in the source language definition. In each expression $\mathsf{aux\text{-}let\text{-}in}(I, E, E')$, we require I to be used at most once in E'.

In order to represent environments, we introduce a notion of R-context:

$$R \;=\; [] \mid \mathsf{apply}(\mathsf{lambda}(I, R), V) \tag{44}$$

We tacitly assume that all bound variables in R are distinct. Each SOS transition specified by

$$\mathsf{env}\ \rho \vdash E \longrightarrow E'$$

can be embedded as

$$R_\rho[E] \longrightarrow R_\rho[E']$$

where, for $\rho = \{I_1 \mapsto V_1, \ldots, I_n \mapsto V_n\}$, $R_\rho = \mathsf{apply}(\mathsf{lambda}(I_n, (\ldots, \mathsf{apply}(\mathsf{lambda}(I_1, []), V_1), \ldots)), V_n)$. We silently assume permutation in R-contexts. We introduce M-contexts to represent the auxiliary environment, in a similar manner to R-contexts, though using $\mathsf{aux\text{-}let\text{-}in}$. In order to represent signals, we extend this notion to one of S-context, introducing a new ternary value constructor ctrl, which is not part of the expression definition but only of the RS representation of SOS.

$$M \;=\; [] \mid \mathsf{aux\text{-}let\text{-}in}(I, V, M) \tag{45}$$

$$S \;=\; M \mid \mathsf{ctrl}(V, I, M) \tag{46}$$

SOS transitions specified by

A) $\mathsf{aux\text{-}env}\ \mu, \mathsf{env}\ \rho \vdash E \xrightarrow{\mathsf{ctrl}\ \mathbf{none}} E'$ B) $\mathsf{aux\text{-}env}\ \mu, \mathsf{env}\ \rho \vdash E \xrightarrow{\mathsf{ctrl}\ \mathbf{some}(F,I)} E'$

can be represented, respectively, as

A) $M_\mu[R_\rho[E]] \longrightarrow M_\mu[R_\rho[E']]$ B) $M_\mu[R_\rho[E]] \longrightarrow \mathsf{ctrl}(F, I, M_\mu[R_\rho[E']])$

where, for $\mu = \{I_1 \mapsto V_1, \ldots, I_n \mapsto V_n\}$, $M_\mu = \mathsf{aux\text{-}let\text{-}in}(I_n, V_n, \ldots \mathsf{aux\text{-}let\text{-}in}(I_1, V_1, []))$. We assume that S-bound variables are distinct from each other and from all the R-bound ones. As with R-contexts, we silently assume permutation for M-contexts.

In this way, we define a one-to-one translation relation $\mathsf{T}(\_,\_)$ between SOS configurations and LS expressions. Applying the translation to the SOS rules gives us the LS models for λV and λDC (resp. LS-V and LS-DC). In particular, the rule for bv, which uses the environment, can be expressed as follows:

$$S[\mathsf{apply}(\mathsf{lambda}(I, R[\mathsf{bv}(I)]), V)] \longrightarrow S[\mathsf{apply}(\mathsf{lambda}(I, R[V]), V)] \tag{47}$$

The aux-bv rule, which uses the auxiliary environment, takes the following form:

$$\mathsf{aux\text{-}let\text{-}in}(I, V, M[R[\mathsf{aux\text{-}bv}(I)]]) \longrightarrow \mathsf{aux\text{-}let\text{-}in}(I, V, M[R[V]]) \tag{48}$$

The LS model for λDC extends LS-V with three more rules: the SOS rule for control,

$$\frac{\mathit{val}(F) \qquad \mathit{fresh\text{-}id}(I)}{M[R[\mathsf{control}(F)]] \longrightarrow \mathsf{ctrl}(F, I, M[R[\mathsf{aux\text{-}bv}(I)]])} \tag{49}$$

an additional congruence rule (which can only match Rule 49),

$$\frac{M[R[E]] \longrightarrow \mathsf{ctrl}(F, I, M[R[E']])}{M[R[P[E]]] \longrightarrow \mathsf{ctrl}(F, I, M[R[P[E']]])} \tag{50}$$

and an encoding of the SOS rule for control-in-prompt (Rule 23).

$$\frac{M[R[E]] \longrightarrow \mathsf{ctrl}(F, I', M[R[E']]) \qquad K = \mathsf{lambda}(I, \mathsf{aux\text{-}let\text{-}in}(I', \mathsf{bv}(I), E')) \qquad \mathit{fresh\text{-}id}(I)}{M[R[\mathsf{prompt}(E)]] \longrightarrow M[R[\mathsf{prompt}(\mathsf{apply}(F, K)]]} \tag{51}$$

Since there is a one-to-one correspondence between LS and SOS transitions (treating R- and M-permutations as silent transitions), and taking for simplicity the identity relation modulo reordering of the environments as observational equivalence in SOS, the following is straightforward.

Prop. 1 The LS model of λV and the corresponding SOS one are derivationally equivalent and adequate with respect to the translation T, and similarly for the LS and SOS models of λDC.

Proof: First we prove derivational equivalence, which is straightforward for λV. The LS control rules correspond to the SOS ones, given the representation of the auxiliary environments and signals. Adequacy follows as the definition of value is the same in all these systems.

5.3 Lifting RC to LR

To facilitate comparison with LS, we define LR as an environment-based version of RC, using R- and M-contexts to represent environments as in LS, and also extend the language with aux-let-in and closure. In the LR models, the reduction rules can be specified as in RC, relying on evaluation contexts. For the way aux-let-in and closure are used, no change is needed in the definition of context. However, since reduction steps now have to be lifted by R- and M-contexts, we replace the single context propagation rule that sufficed in RC with the four following rules: *lifting*, *lifted congruence* and R- and M-*lowering*.

$$\frac{E \longrightarrow E'}{M[R[E]] \longrightarrow M[R[E']]} \tag{52}$$

$$\frac{M[R[E]] \longrightarrow M[R[E']]}{M[R[Q[E]]] \longrightarrow M[R[Q[E']]]} \tag{53}$$

$$\frac{M[\mathsf{apply}(\mathsf{lambda}(I,R[E]),V)] \longrightarrow M[\mathsf{apply}(\mathsf{lambda}(I,R[E']),V)]}{M[R[\mathsf{apply}(\mathsf{lambda}(I,E),V)]] \longrightarrow M[R[\mathsf{apply}(\mathsf{lambda}(I,E'),V)]]} \tag{54}$$

$$\frac{\mathsf{aux\text{-}let\text{-}in}(I,V,M[R[E]]) \longrightarrow \mathsf{aux\text{-}let\text{-}in}(I,V,M[R[E']])}{M[R[\mathsf{aux\text{-}let\text{-}in}(I,V,E)]] \longrightarrow M[R[\mathsf{aux\text{-}let\text{-}in}(I,V,E')]]} \tag{55}$$

We assume that LR models include R- and M-permutations, as well as the bv and aux-bv rules, and Rule 50. Notice that evaluation can also apply to open terms, however we do not need to change our definition of value, as substitution of free variables is dealt with as in LS, by the bv rule. We also need the following rule to deal with closures:

$$\frac{\rho = \{I_1 \mapsto V_1, \ldots I_n \mapsto V_n\} \qquad \mathit{free\text{-}vars}(E) \subseteq \{I_1,\ldots,I_n\}}{\mathsf{closure}(\rho,I,E) \longrightarrow \mathsf{apply}(\mathsf{lambda}(I_n,(\ldots,\mathsf{apply}(\mathsf{lambda}(I_1,\mathsf{lambda}(I,E)),V_1),\ldots)),V_n)} \tag{56}$$

This gives us the LR model of λV (LR-V). We can extend this model with rules for delimited control in two ways: to simulate RC (LR-DC), or to simulate SOS (LX-DC). The LR-DC rules for prompt and control are those based on the RC one (i.e. they are the lifted version of Rules 42 and 43), and they do not involve any use of the auxiliary notation. LR-DC is the extension of LR-V with these rules. On the other hand, the LX-DC model is obtained by extending LR-V with the LS control rules (Rules 49 and 51, which rely on the auxiliary notation).

The following can be proved for all the systems we are considering, i.e. with respect to $\equiv_X$ where $X \in \{\text{LS-V}, \text{LR-V}, \text{LS-DC}, \text{LX-DC}, \text{LR-DC}\}$.

Prop. 2 A) $V \in \mathit{Val}_{\text{SOS}}$ whenever there exists $V' \in \mathit{Val}_{\text{RC}}$ such that $V \equiv_X V'$.

B) $S[R_\rho[\mathsf{apply}(\mathsf{lambda}(I,E),V_1)]] \longrightarrow_{\text{LS}} V_2$ with $V_1, V_2 \in \mathit{Val}_{\text{SOS}}$ whenever there exist $V_3, V_4 \in \mathit{Val}_{\text{RC}}$ such that $S[[\mathsf{apply}(\mathsf{closure}(\rho,I,E),V_3)]] \longrightarrow_{\text{LR}} V_4$, with $V_1 \equiv_X V_3$ and $V_2 \equiv_X V_4$.

The following provable equivalences correspond respectively to the bv rule, to β-reduction, and to aux-let-in elimination, for $\equiv_X$ as before:

$$\mathsf{apply}(\mathsf{lambda}(I,\mathsf{bv}(I)),V) \equiv_X \mathsf{apply}(\mathsf{lambda}(I,V),V) \tag{57}$$

$$\mathsf{apply}(\mathsf{lambda}(I,E),V) \equiv_X E\{\mathsf{bv}(I) \mapsto V\} \tag{58}$$

$$\mathsf{aux\text{-}let\text{-}in}(I,V,E) \equiv_X E\{\mathsf{aux\text{-}bv}(I) \mapsto V\} \tag{59}$$

The following can now be proved:

Prop. 3 $\mathsf{apply}(\mathsf{lambda}(I,\mathsf{aux\text{-}let\text{-}in}(I',\mathsf{bv}(I),P[\mathsf{aux\text{-}bv}(I')])),V) \equiv_X \mathsf{apply}(\mathsf{lambda}(I,P[\mathsf{bv}(I)]),V)$

Proof: $\mathsf{apply}(\mathsf{lambda}(I,\mathsf{aux\text{-}let\text{-}in}(I',\mathsf{bv}(I),P[\mathsf{aux\text{-}bv}(I')])),V) \equiv_X \mathsf{aux\text{-}let\text{-}in}(I',V,P[\mathsf{aux\text{-}bv}(I')])$, by Equiv. 58.

$\mathsf{aux\text{-}let\text{-}in}(I',V,P[\mathsf{aux\text{-}bv}(I')]) \equiv_X P[V]$, by Equiv. 59, observing that $\mathsf{aux\text{-}bv}(I')$ cannot occur free in P, as it must be used at most once in $P[\mathsf{aux\text{-}bv}(I')]$.

$P[V] \equiv_X \mathsf{apply}(\mathsf{lambda}(I,P[\mathsf{bv}(I)]),V)$, by Equiv. 58.

The following is an immediate consequence of Prop. 3, applying functional extensionality.

$$\mathsf{lambda}(I,\mathsf{aux\text{-}let\text{-}in}(I',\mathsf{bv}(I),P[\mathsf{aux\text{-}bv}(I')])) \equiv_X \mathsf{lambda}(I,P[\mathsf{bv}(I)]) \tag{60}$$

5.4 Adequacy of SOS and RC

We first show that the LR models and the RC ones are equivalent.

Prop. 4 LR-V and RC-V are adequate, and so too are LR-DC and RC-DC.

Proof: The language of RC is included in that of LR, hence we can take the identity on RC (denoted by $\mathsf{Id}_{\mathrm{RC}}$) as the translation. The LR models can be obtained by a lambda-lifting refactoring of the RC models, and this gives equivalent systems, under a change of the evaluation strategy that affects only the top level. We prove derivational equivalence by induction on the structure of derivations, relying on Equiv. 59 and Rule 56 to eliminate the additional LR syntax, observing that aux-let-in and closure are inessential in LR-DC (they can only be eliminated without leading to any new values). Adequacy follows immediately as values are defined in the same way in the two systems.

We consider the relationship between the different representations of λV.

Prop. 5 The LS model of λV and the corresponding LR one are adequate.

Proof: The two models use the same language, hence we can take the identity translation. They differ on congruence rules and reduction of function application. Congruence rules in LR are expressed using CBV-contexts, unlike in LS, but both are equivalent specifications of CBV. For equivalence with respect to values and function application, we rely on Prop. 2.

We extend this result to SOS-style delimited control.

Prop. 6 LS-DC and LX-DC are adequate.

Proof: We first prove derivational equivalence with respect to identity using Prop. 5 and the fact that the two extensions are obtained by adding the same rules.

We finally compare SOS-style and RC-style delimited control.

Prop. 7 LX-DC and LR-DC are derivationally equivalent and adequate.

Proof: We prove derivational equivalence by induction on the structure of derivations, with respect to the identity translation. The two systems are equivalent up to λV by Prop. 4, so the only interesting case is delimited control, in which respect LX-DC stepwise behaves as LS-DC. The lifted version of the RC prompt rule (Rule 42) is in both systems. Rules 49 and 50 can be added to LR-DC without expanding the set of derivable values. Thus the only possible difference between the two systems is between the natural LR control rule (the lifted version of Rule 43) of LR-DC, and the LS control-in-prompt rule (Rule 51) of LX-DC. We now show that the two rules are interderivable (i.e. given the system with one rule, the other one is derivable). First we observe that, by Eq. 60, the specification of the continuation K in either rule is equivalent to that in the other, and therefore interchangeable.

From S to R: in order to derive the LR-DC rule from the LX-DC one, we observe that a lifted expression $M[R[P[\mathsf{control}(F)]]]$, where F is a value, can be reduced to $\mathsf{ctrl}(F,I,M[R[P[\mathsf{aux\text{-}bv}(I)]]])$ in LX-DC, using the control rule (Rule 49), and the applicable congruence rule (Rule 50). This gives us the premise for the application of the LX-DC control-in-prompt rule to $M[R[\mathsf{prompt}(P[\mathsf{control}(F)])]]$ in a way that simulates the LR-DC control rule.

From R to S: in LX-DC (as in LS-DC) a one-step transition from $M[R[E]]$ to $\mathsf{ctrl}(F,I,M[R[E']])$ is only possible provided $E \equiv P[\mathsf{control}(F)]$ and $E' \equiv P[\mathsf{aux\text{-}bv}(I)]$ for some P (possibly relying on the conversion of closures to function applications). Therefore, the LR-DC control rule can be applied to $M[R[\mathsf{prompt}(E)]]$ to simulate Rule 51.

Diagramatically, the overall proof can be presented as follows (where vertical arrows denote model inclusion).

$$\begin{array}{ccccccccc} \text{SOS-DC} & \sim^{\mathsf{T}} & \text{LS-DC} & \sim^{\mathsf{Id}} & \text{LX-DC} & \sim^{\mathsf{Id}} & \text{LR-DC} & \sim^{\mathsf{Id}_{\text{RC}}} & \text{RC-DC} \\ \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ \text{SOS-V} & \sim^{\mathsf{T}} & \text{LS-V} & \sim^{\mathsf{Id}} & \text{LR-V} & = & \text{LR-V} & \sim^{\mathsf{Id}_{\text{RC}}} & \text{RC-V} \end{array}$$

Prop. 8 SOS-DC and RC-DC are derivationally equivalent and adequate with respect to $\mathsf{T} \circ \mathsf{Id}_{\text{RC}}$

Proof: based on Props. 1, 4, 6, 7, using the fact that adequacy is transitive, by composition of the translation relations, and by transitivity of observational equivalence.

6 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. [18, 22, 23, 24, 30, 33]) 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) [12, 16, 31, 33], or a lower-level operational specification by translation to abstract-machine code [6, 19]. At a higher level, algebraic characterisations of control operators have been given in terms of equational theories [18, 23].

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 [13] suffices to express *call/cc*, *shift* and *reset*; however, more sophisticated CPS transformations are needed to express *control* and *prompt* [33].

Felleisen's [19] 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* [18], where continuations are represented as evaluation contexts and exchange rules are not needed. Felleisen [19] 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 [12, 13]. Continuations were treated as functions, relying on the meta-continuation approach [12] 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 by Danvy and Yang [15], and a specification based on evaluation contexts has been given by Kameyama and Hasegawa [23], together with an algebraic characterisation.

Giving a CPS semantics to *control* is significantly more complex than for *shift* [33]. 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 [18], the monadic framework in [16], and the operational framework in [6]. Relying on the introduction of recursive continuations, Shan [33] 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 [18, 23, 24, 33].

As shown by Filinski [20], *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 [33]. A direct implementation of *control* and *shift* has been given by Gasbichler and Sperber [21]. A CPS-based implementation of control operators in a monadic framework has been given by Dyvbig et al [16]. A semantics of *call/cc* based on an efficient implementation of evaluation contexts is provided in the K Framework [30].

7 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. Definitions based on evaluation contexts are often even more concise than the corresponding MSOS definitions, since a single alternative in a context-free grammar for evaluation contexts subsumes an entire MSOS rule allowing evaluation of a particular subexpression. However, such grammars are significantly less modular than MSOS rules: adding a new control operator to a specified language may require duplication of a (potentially large) grammar [17, e.g. pages 141–142]. (This inherent lack of modularity of evaluation context grammars is addressed in the PLT Redex tools by the use of ellipsis.)

We initially validated our specifications through a suite of 70 test programs, which we accumulated from examples in the literature on control operators ([1, 2, 4, 6, 9, 10, 12, 18, 19, 22, 33]). 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 [32].

While the test programs demonstrated that we had successfully specified a control operator that behaves very similarly to the operator *control* described in the literature, they did not prove that we had specified exactly the same operator. We addressed this in Section 5, where we proved that our MSOS specification is equivalent to a conventional specification using a reduction semantics based on evaluation contexts (e.g. [18, 24]).

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 PLAN-COMPS project and by EU funding (Horizon 2020, grant 640954) to KU Leuven for the GRACEFUL 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 & M. Wand (1998): *Revised^5 Report on the Algorithmic Language Scheme*. *Higher-Order and Symbolic Computation* 11(1), pp. 7–105, doi:10.1023/A:1010051815785.

[2] Kenichi Asai & Yukiyoshi Kameyama (2007): *Polymorphic Delimited Continuations*. In: *Asian Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science* 4807, Springer, pp. 239–254, doi:10.1007/978-3-540-76637-7_16.

[3] Egidio Astesiano (1991): *Inductive and Operational Semantics*. In: *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, Springer, pp. 51–136. ISBN 978-3-540-53961-2.

[4] Malgorzata Biernacka, Dariusz Biernacki & Olivier Danvy (2005): *An Operational Foundation for Delimited Continuations in the CPS Hierarchy*. *Logical Methods in Computer Science* 1(2), pp. 1–39, doi:10.2168/LMCS-1(2:5)2005.

[5] Dariusz Biernacki & Olivier Danvy (2006): *A Simple Proof of a Folklore Theorem about Delimited Control*. *Journal of Functional Programming* 16(3), pp. 269–280, doi:10.1017/S0956796805005782.

[6] Dariusz Biernacki, Olivier Danvy & Chung-chieh Shan (2006): *On the Static and Dynamic Extents of Delimited Continuations*. *Science of Computer Programming* 60(3), pp. 274–297, doi:10.1016/j.scico.2006.01.002.

[7] Martin Churchill & Peter D. Mosses (2013): *Modular Bisimulation Theory for Computations and Values*. In: *International Conference on Foundations of Software Science and Computation Structures*, *Lecture Notes in Computer Science* 7794, Springer, pp. 97–112, doi:10.1007/978-3-642-37075-5_7.

[8] Martin Churchill, Peter D. Mosses, Neil Sculthorpe & Paolo Torrini (2015): *Reusable Components of Semantic Specifications*. In: *Transactions on Aspect-Oriented Software Development XII*, *Lecture Notes in Computer Science* 8989, Springer, pp. 132–179, doi:10.1007/978-3-662-46734-3_4.

[9] William Clinger (1987): *The Scheme Environment: Continuations*. *SIGPLAN Lisp Pointers* 1(2), pp. 22–28, doi:10.1145/1317193.1317197.

[10] Olivier Danvy (2006): *An Analytical Approach to Programs as Data Objects*. DSc thesis, Department of Computer Science, Aarhus University. Available at `http://cs.au.dk/~danvy/DSc/`.

[11] Olivier Danvy (2008): *Defunctionalized Interpreters for Programming Languages*. In: *International Conference on Functional Programming*, ACM, pp. 131–142, doi:10.1145/1411204.1411206.

[12] Olivier Danvy & Andrzej Filinski (1989): *A Functional Abstraction of Typed Contexts*. Technical Report 89/12, DIKU, University of Copenhagen. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.4822`.

[13] Olivier Danvy & Andrzej Filinski (1990): *Abstracting Control*. In: *Conference on LISP and Functional Programming*, ACM, pp. 151–160, doi:10.1145/91556.91622.

[14] Olivier Danvy & Andrzej Filinski (1992): *Representing Control: A Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2(4), pp. 361–391, doi:10.1017/S0960129500001535.

[15] Olivier Danvy & Zhe Yang (1999): *An Operational Investigation of the CPS Hierarchy*. In: *European Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science* 1576, Springer, pp. 224–242, doi:10.1007/3-540-49099-X_15.

[16] R. Kent Dyvbig, Simon Peyton Jones & Amr Sabry (2007): *A Monadic Framework for Delimited Continuations*. *Journal of Functional Programming* 17(6), pp. 687–730, doi:10.1017/S0956796807006259.

[17] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics Engineering with PLT Redex*. MIT Press. ISBN 9780262062756.

[18] Matthias Felleisen, Mitch Wand, Daniel Friedman & Bruce Duba (1988): *Abstract Continuations: A Mathematical Semantics for Handling Full Jumps*. In: *Conference on LISP and Functional Programming*, ACM, pp. 52–62, doi:10.1145/62678.62684.

[19] Mattias Felleisen (1988): *The Theory and Practice of First-Class Prompts*. In: *Symposium on Principles of Programming Languages*, ACM, pp. 180–190, doi:10.1145/73560.73576.

[20] Andrzej Filinski (1994): *Representing Monads*. In: *Symposium on Principles of Programming Languages*, ACM, pp. 446–457, doi:10.1145/174675.178047.

[21] Martin Gasbichler & Michael Sperber (2002): *Final Shift for Call/cc: Direct Implementation of Shift and Reset*. In: *International Conference on Functional Programming*, ACM, pp. 271–282, doi:10.1145/581478.581504.

[22] Carl A. Gunter, Didier Rémy & Jon G. Riecke (1995): *A Generalization of Exceptions and Control in ML-like Languages*. In: *International Conference on Functional Programming Languages and Computer Architecture*, ACM, pp. 12–23, doi:10.1145/224164.224173.

[23] Yukiyoshi Kameyama & Masahito Hasegawa (2003): *A Sound and Complete Axiomatization of Delimited Continuations*. In: *International Conference on Functional Programming*, ACM, pp. 177–188, doi:10.1145/944705.944722.

[24] Yukiyoshi Kameyama & Takuo Yonezawa (2008): *Typed Dynamic Control Operators for Delimited Continuations*. In: *International Symposium on Functional and Logic Programming*, *Lecture Notes in Computer Science* 4989, Springer, pp. 239–254, doi:10.1007/978-3-540-78969-7_18.

[25] Peter D. Mosses (2002): *Pragmatics of Modular SOS*. In: *International Conference on Algebraic Methodology and Software Technology*, *Lecture Notes in Computer Science* 2422, Springer, pp. 21–40, doi:10.1007/3-540-45719-4_3.

[26] Peter D. Mosses (2004): *Modular Structural Operational Semantics*. *Journal of Logic and Algebraic Programming* 60–61, pp. 195–228, doi:10.1016/j.jlap.2004.03.008.

[27] Peter D. Mosses & Mark J. New (2009): *Implicit Propagation in Structural Operational Semantics*. In: *Workshop on Structural Operational Semantics*, *Electronic Notes in Theoretical Computer Science* 229(4), Elsevier, pp. 49–66, doi:10.1016/j.entcs.2009.07.073.

[28] Peter D. Mosses & Ferdinand Vesely (2014): *FunKons: Component-Based Semantics in K*. In: *International Workshop on Rewriting Logic and Its Applications*, *Lecture Notes in Computer Science* 8663, Springer, pp. 213–229, doi:10.1007/978-3-319-12904-4_12.

[29] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics*. *Journal of Logic and Algebraic Programming* 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001. Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[30] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the K Semantic Framework*. *Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[31] Amr Sabry & Matthias Felleisen (1993): *Reasoning About Programs in Continuation-passing Style*. *LISP and Symbolic Computation* 6(3–4), pp. 289–360, doi:10.1007/BF01019462.

[32] Neil Sculthorpe, Paolo Torrini & Peter D. Mosses (2016): *A Modular Structural Operational Semantics for Delimited Continuations: Additional Material*. Available at `http://www.plancomps.org/woc2016`.

[33] Chung-chieh Shan (2007): *A Static Simulation of Dynamic Delimited Control*. *Higher-Order and Symbolic Computation* 20(4), pp. 371–401, doi:10.1007/s10990-007-9010-4.

[34] Dorai Sitaram & Matthias Felleisen (1990): *Control Delimiters and their Hierarchies*. *LISP and Symbolic Computation* 3(1), pp. 67–99, doi:10.1007/BF01806126.