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

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

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

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

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

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

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

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

$$\frac{\rho(I) = V}{\textsf{env}\ \ \rho \vdash \textsf{\textbf{bv}}(I) \rightarrow V} \tag{7}$$

$$\textsf{env}\ \ \rho \vdash \textsf{\textbf{lambda}}(I, E) \rightarrow \textsf{\textbf{closure}}(\rho, I, E) \tag{8}$$

$$\frac{E_1 \rightarrow E_1'}{\textsf{\textbf{apply}}(E_1, E_2) \rightarrow \textsf{\textbf{apply}}(E_1', E_2)} \tag{9}$$

$$\frac{val(V) \qquad E \rightarrow E'}{\textsf{\textbf{apply}}(V, E) \rightarrow \textsf{\textbf{apply}}(V, E')} \tag{10}$$

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

$$\frac{val(V_1) \qquad val(V_2)}{\textsf{\textbf{apply}}(\textsf{\textbf{closure}}(\rho, I, V_1), V_2) \rightarrow V_1} \tag{12}$$

Figure 2: I-MSOS rules for lambda calculus.

absence of an exception, or **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:

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

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

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.

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

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

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

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

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 **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 **meta-bv**(I), which looks up the identifier I in the meta-environment, and **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:

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

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

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, in contrast to the preliminary Rule 13, uses **meta-bv** to lookup I in the meta-environment.

We then specify **prompt** as follows:

$$\frac{\mathit{val}(V)}{\textbf{prompt}(V) \rightarrow V} \tag{21}$$

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

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

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:

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

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

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**:

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

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

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

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

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

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**:

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

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:

$$\frac{E \rightarrow E'}{\textsf{abort}(E) \rightarrow \textsf{abort}(E')} \tag{31}$$

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

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):

$$\frac{E \rightarrow E'}{\textsf{callcc}(E) \rightarrow \textsf{callcc}(E')} \tag{33}$$

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

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