An Equational Specification for the Scheme Language

Marcelo d'Amorim and Grigore Roșu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL, 61801, USA
{damorim, grosu}@uiuc.edu

Abstract. This work describes the formal semantics of Scheme as an equational theory in the Maude rewriting system. The semantics is based on continuations and is highly modular. We briefly investigate the relationship between our methodology for defining programming languages and other semantic formalisms. We conclude by showing some performance results of the interpreter obtained for free from the executable specification.

1 Introduction

Following the approach proposed in [24] and partially published in [18] to give modular rewriting logic [16] executable semantics to programming languages, this paper describes a formal semantics of Scheme. The efficient rewriting engine of MAUDE [3] gives us an interpreter for Scheme from its semantics essentially for free. Since Scheme is not a concurrent language, we only need the equational fragment of rewriting logic.

The use of rewriting logic is not novel in the definition of programming languages. In fact, one can translate the structural operational semantics (SOS) [23] of a sequential language into a set of equations [18, 17] satisfying the Church-Rosser property. Braga et al. [5] investigated how to translate language definitions given in the form of Mosses’ MSOS [21] into modular rewriting semantics (MRS) [6] in order to mechanically derive sound language interpreters for concurrent languages.

As in [5], our main focus is on modular definitions of languages using rewriting. In [24], we found (and empirically evaluated in class) that the use of continuations [4, 10], in addition to that of associative and commutative (AC) sets of state attributes, can improve modularity of language definitions. Using this technique, we here illustrate how easily a complex language like Scheme can be defined. In particular, we show how the Scheme construct call/cc is precisely defined with only 3 equations. This methodology has been used for the last two years in a programming language design class taught by the second author at the University of Illinois [24]. Even though Scheme is not concurrent in its core, the methodology we present also supports concurrency, as shown in [24, 18].

Section 2 describes related work. Section 3 introduces informally the Maude system and the Scheme language. Section 5 shows the executable semantics of Scheme. Section 5 illustrates performance results and Section 6 concludes the paper.

2 Related Work

Monads have been introduced [26] as a data type allowing one to add “impure” features, such as side-effects, to pure functional languages. They became notable for their contribution to modularity. Monads can be seen as a structuring feature that enables one to systematically write programs in tail-form. With monads it is easier to localize changes since context is transferred in a uniform manner. Mainly because of the use of continuations, transitions in our semantics are defined in a similar fashion as the monad operator *. Matching modulo ACI allows one to define * via equations with some degree of freedom.

MSOS (Modular Structural Operational Semantics) [21] is a formalism which aims at describing modular definitions of languages. The technique in [21] proposes to overcome the possible lack of modularity in

1 Based on the publicly available report R5RS [15].
standard SOS [23] specifications. In MSOS, transitions take the form $\gamma \xrightarrow{\alpha} \gamma'$, where $\gamma$ and $\gamma'$ are called configurations and denote syntactic entities of the language (expression and commands, for instance). One can understand these transitions as describing source-to-source derivations rules. The label $\alpha$ transmits only the necessary information needed in a particular derivation, such as the store and the environment. Modularity is obtained essentially by avoiding the inclusion of auxiliary information explicitly in configuration terms. It is worth noting that MSOS [21, page 221] was influenced by monad transformers.

Braga et al. [7,5,6] investigated modularity of rewrite theories for defining programming languages. Meseguer and Braga [17] recently illustrated a technique, named Modular Rewriting Semantics (MRS) [17], for defining modular rewrite theories. They propose the use of record inheritance, and abstract interfaces in order to give definitive formalizations of language constructs via conservative extensions of the semantics. The first technique is similar to the notion of set of state attributes we use, which is essentially an AC list of attributes representing the “state of the program”. The store, environment, and locks held by each thread are candidate attributes for this list in the semantics of concurrent languages. Matching modulo ACI makes possible to specify equations containing only a projection of the attributes in the list. Similarly to MSOS labels and MRS record inheritance, the use of sets of state attributes allows one to define only what is needed in order to carry out a derivation step, thus reducing the coupling between infrastructure and semantic definitions. The use of abstract interfaces as the second technique follows the information hiding discipline [22] which is a fundamental concept to achieve modularity. In our semantics, this boils down to defining separate modules for the syntax and semantics of each construct, and accessing modules via their public operations.

It is worth mentioning that in this paper we do not aim at formalizing our definitional methodology, or at comparing it mathematically to other existing techniques. Instead, our purpose here is to instantiate it to one non-trivial special case, the definition of SCHEME. We believe that the presentation will be self-explanatory and intuitive enough to make this possible.

3 Preliminaries

In this section we present a short introduction to the MAUDE language, then give an informal semantics to some SCHEME constructs.

MAUDE is a high-performance executable specification language for rewriting logics whose roots go back to Clear and OBJ. Many other languages belong to the same family, such as CAFEOBJ, BOBJ, and ELAN.

In MAUDE, specifications are introduced as theory modules. In this paper we only use functional modules, and for these, specifications correspond to equational theories of the form $(\Sigma, E)$, where $\Sigma$ denotes the signature of a module including its sorts and operations defining the interface to that module, and $E$ denotes the set of equations that should hold for any implementation of that module. MAUDE replaces terms by terms when they are equal under an equational theory. That is, for any equation of the form $t_l = t_r$, when a term $t$ can be matched modulo associativity, commutativity, and identity (ACI) to $t_l$ via a substitution $\theta$, then $t$ can be replaced by $t_r$ with variables substituted according to $\theta$. As an example, the module below axiomatizes the PEANO natural numbers:

```
mod PEANO-NAT is sort Nat .
  vars N M : Nat . eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endfm
```

The only sort defined in this module is Nat. The constant zero is defined as an operation with no arguments and has sort Nat. The operations succ and plus denote the successor of a number and the addition of two numbers, respectively. The equations in this module must hold for any model of PEANO-NAT. In these equations, variables are universally quantified. That is, the first equation describes the axiom: $\forall M: \text{Nat}. \ 0 + M = M$. One can prove via equational deduction that the equality $\text{plus}(\text{succ}(\text{succ}(\text{zero})), \text{succ}(\text{succ}(\text{succ}(\text{zero})))) = \text{plus}(\text{succ}(\text{succ}(\text{succ}(\text{zero})))), \text{succ}(\text{zero}))$ holds for this specification since both terms reduce to the ground term: $\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{zero})))))$.

In fact, one can show that this specification is confluent and terminates. One can import a module (theory) using the keyword including. All sorts, operations, and equations are then imported. The module PEANO-NAT* below extends the previous with multiplication.
fmod PEANO-NAT* is including PEANO-NAT .

  op mult : Nat Nat -> Nat .
  vars M N : Nat .
  eq mult(zero, M) = zero .
  eq mult(succ(N), M) = plus(mult(N, M), M) .
endfm

This simple examples illustrate most of MAUDE’s features needed to define SCHEME. We explain other features by need. For a more complete description of MAUDE see [3].

SCHEME [15,?] is a functional programming language mostly inspired by LISP. The following is a non-exhaustive list of its features:

- Statically scoped;
- Call-by-value (but can simulate call-by-need with the Promises data-type);
- Programs as data;
- Functions and continuations are first-class citizens;
- Dynamically typed.

SCHEME has a small core language. We next informally discuss its main constructs. The interested reader is referred to [15] for the standard definition and many examples. Notice in the following that, similarly to LISP, every expression in SCHEME has the syntax of a list.

Lists. The construct list has the form (list exp1 ... expK) and evaluates to a list of size k whose values correspond to the evaluation of each expression argument.

Variables. The binding construct let has the form

(state ((name1 exp1) ... (nameK expK)) exp1' ... expN')

and “atomically” binds the results of evaluating exp1, ..., expK in the current environment to name1, ..., nameK. The expressions exp1', ..., expN' are evaluated sequentially in the new environment obtained. The result of the whole expression is that of the last expression, expN', evaluated in this sequence. Variations of this bindings include let* and letrec. The first performs the binding sequentially rather than atomically, while letrec binds in an environment initially extended with the name list name1, ..., nameK. As usual, the letrec construct is very convenient for defining recursive definitions.

The construct set! has the form (set! name exp) and assigns the evaluation of exp to name in the current environment.

The construct define takes the form (define name exp) and binds the evaluation of exp to name in the top-level environment if name is not yet bound. Otherwise, this expression is similar to an assignment to name. Our interpreter does not allow the use of define outside the top-level block (it can be encoded with letrec [15]).

Functions. The function abstraction construct has the form

(lambda (name1 ... nameK) exp1' ... expN')

and evaluates to a closure. This closure saves the current environment because SCHEME is statically scoped. In addition, since the language supports high-order functions, closures are ordinary values in the language.

Function application takes the form (exp0 exp1 ... expK) where exp0 must evaluate to a closure. Assuming that the closure that exp0 evaluates to holds the function abstraction above and the environment E, then the result of evaluating the application above is the same as that of evaluating

(let ((name1 exp1) ... (nameK expK)) exp1' ... expN')

in E. Recall that SCHEME’s parameter-passing style is by-value.

Control Flow. The if construct has the form (if test then-exp else-exp) and has the expected behavior of evaluating the expression test first and if the result is the “true” value then the result of the conditional is that of evaluating then-exp; otherwise that of evaluating else-exp.

Sequential composition has the form (begin exp1 ... exprK) and evaluates sequentially each expression. The evaluation of exprK is the result of the entire expression.

Continuations are procedures of a single parameter and thus considered first-class citizens in SCHEME. In the example below, k denotes a continuation procedure. The actual parameter passed to the continuation
procedure denotes the value that the program continuation expects to receive in order to continue. The way one creates a continuation procedure is by calling the procedure call/cc passing an expression denoting a function abstraction, which declares one formal standing for the continuation procedure. For instance, the Scheme expression:

\[
\text{(let ((f (lambda (k) (k 10))))(* 5 (call/cc f)))}
\]

evaluates to the integer 50. When the function \(f\) is applied, a continuation procedure (that knows “how to continue”) is created and passed as argument to \(f\). The value 10 is then passed to the program continuation which is in charge of filling the hole \(\square\) in the expression \((\ast 5 \square)\) and evaluate it.

**Data Types.** In addition to function, continuation, and list data-types, Scheme supports integer, rational, real, string, boolean, and several other data-types. We here discuss only the integers and booleans. Integer literals take the form \(1, 2, 3, \ldots\), while the boolean literals are \#t and \#f. Arithmetic operations on these types are provided as built-in operations. For instance, \((+ 1 2)\) denotes the addition of two integers and \((\text{and} a b)\) denotes the conjunction of two expressions.

Scheme also supports “frozen” expressions as values: expressions may evaluate to “frozen” expressions, also called “data”. With the support of constructs to freeze and unfreeze “data”, one can store programs as data. For instance, the construction \((\text{quote exp})\) freezes the expression \(\text{exp}\). That means that \(\text{exp}\) is not evaluated when \((\text{quote exp})\) is. Conversely, the expression \((\text{unquote exp})\) evaluates the frozen expression that can be resulted from the evaluation of \(\text{exp}\) in the current environment.

Promises are introduced to support call-by-need. Two operations support this feature: \((\text{delay exp})\) and \((\text{force exp})\). Similarly to quotations, the evaluation of \(\text{delay}\) does not trigger the evaluation of \(\text{exp}\). It saves the current environment and \(\text{exp}\) in a promise data-type. When forced, the promise will evaluate \(\text{exp}\) in the saved environment. Additional calls to \(\text{force}\) on the same promise will return the same value. As one might expect, in the presence of side-effects, the evaluation of an expression can differ from an eventual force in the promise corresponding to that expression.

## 4 The SchemeM Syntax

We use a slightly different syntax in order to facilitate parsing and semantic definitions. We call this language SchemeM from here on. We write the syntax of SchemeM in Maude’s mixfix notation which is equivalent to defining a context-free grammar. We next show fragments of the SchemeM syntax. The entire syntax appears in Appendix A. We decided to have a single sort for expressions, namely Exp, in order to simplify the definition and parsing. The sorts, subsort relations, and operations below define the basic syntax for integers, booleans, and expression lists. The sorts Qid, standing for quoted identifiers, and Int are pre-defined by Maude. NameList is defined in a similar manner as ExpList. The subsorts keyword allows one to declare subsort relations succintely. For instance, we declare Qid to be subsort of Name, and this to be a subsort of Exp:

- **sorts:** Exp ExpList . subsorts Qid < Name Int < Exp NameList < ExpList .
- **subsort Name < NameList .
- op a b c d e f g h i j k l m n o p q r s t u v x y z : -> Name .

The operations plus, minus, times, div, and mod have signature Exp Exp -> Exp. Boolean expressions and the conditional are defined with the following syntax:

- ops true false : -> Exp .
- ops eq leq geq and or : Exp Exp -> Exp . op not : Exp -> Exp .
- op if_then_else_ : Exp Exp Exp -> Exp .

The syntax of letrec follows. The attribute prec assigns precedence to an operator in order to deal with parsing conflicts:

- **sorts Binding BindingList . subsort Binding < BindingList .
- op none : -> BindingList .
- op _,_ : Name Exp -> Binding [prec 70] .
- op letrec_in_ : BindingList Exp -> Exp .

4
The syntax of function abstraction and application is:

\[
\text{op \text{lambda} _ -> _ : NameList Exp -> Exp}
\]
\[
\text{op __ : Exp ExpList -> Exp [prec 0].}
\]

We define next the syntax for quotation, unquotation, callcc, and for the operations on the “promises” data type.

\[
\text{op quote : Exp -> Exp . op unquote : Exp -> Exp .}
\]
\[
\text{op callcc_ : Exp -> Exp [prec 0] .}
\]
\[
\text{op delay : Exp -> Exp . op force : Exp -> Exp .}
\]

Finally, we give the syntax for assignment, sequential composition, and expression blocks. Note that the sort \text{ExpList}; is used to stand out expressions separated by the ; symbol.

\[
\text{op set=_ : Name Exp -> Exp .}
\]
\[
\text{sort ExpList; . subsort Exp < ExpList; .}
\]
\[
\text{op _;_: ExpList; ExpList; -> ExpList; [assoc prec 100] . op {_} : ExpList; -> Exp .}
\]

We implemented, but omitted in this paper for space reasons, the syntactic constructs for: vectors, lists, loops, \text{let*}, \text{let}, and IO operations.

5 The Equational Semantics

Next we show the state infrastructure used in the semantics, describe the continuation semantics, and finally instantiate this semantics to \text{SCHEME\text{M}}.

5.1 The State Infrastructure

In what follows we axiomatize the standard notions of location, environment, value, and store. Each module denotes one relevant component of the state infrastructure, allowing us to eventually define the module \text{STATE}. We start with the formal definition of locations (term of sort \text{Location}), in the module \text{LOCATION} below. The sort of natural numbers, \text{Nat}, is imported from the built-in module \text{INT}. A list of locations can be constructed using the comma mixfix operator \_,\_. In MADE, one can assign certain properties to operators via a fixed set of attributes. For instance, we state that the operator \_,\_ is commutative and \text{noLoc} is its identity. So the term \((\text{loc}(1),\text{noLoc}),\text{loc}(2))\) is equivalent to \(\text{loc}(1),\text{loc}(2)\). It is sometimes convenient to generate a fresh sequence of contiguous locations. This can be done with the operator \text{loc}(_;;_\,) taking as arguments two natural numbers: the index of the first location and the number of locations to create:

fmod \text{LOCATION} is including \text{INT}.

\[
\text{sorts Location LocationList . subsort Location < LocationList .}
\]
\[
\text{op loc : Nat -> Location . op noLoc : -> LocationList .}
\]
\[
\text{op _,_: LocationList LocationList -> LocationList [assoc id: noLoc] .}
\]
\[
\text{op loc : Nat -> Location . op loc(_;;_;_) : Nat Nat -> LocationList .}
\]
\[
\text{vars N # : Nat . eq loc(N ;; 0) = noLoc .}
\]
\[
\text{eq loc(N ;; #) = loc(N), loc((N + 1) ;; (# - 1)) .}
\]
endfm

Next we describe environments. The module \text{NAME\text{-LIST}} defines the sorts \text{Name} and \text{NameList} standing for the language identifier and its associated list (see Appendix A). An environment is a table mapping names to locations. An environment cell has the form \([N,L]\) where \(N\) is a name and \(L\) a location. The operation \_\_ concatenates two environments. Note that, in addition to using the \text{assoc} and \text{id} attributes, we also declare this operator commutative (\text{comm}). It is worth noting that these attributes are central to rewriting, since matching is performed modulo ACI (associativity, commutativity and identity). The operation \text{E[N1<-L1]} updates the environment \(E\) by binding the location list \(L1\) to the name list \(N1\) element-wise.
fmod ENVIRONMENT is including LOCATION . including NAME-LIST .
  sort Env .  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op ___ : Env Env -> Env [assoc comm id: noEnv] .
  op [_,<-_] : Env NameList LocationList -> Env .
  vars X X' : Name .  vars Env Env' : Env .  vars L L' : Location .
  var XL : NameList .  var LL : LocationList .
  eq Env[(X,L) <- noLoc] = Env .
  eq (Env[X,XL <- L',LL] = (Env[X',L'] Env)[X1 <- LL] .
  eq Env[X,X1 <- L,LL] = (Env [X,L])[X1 <- LL] [owise] .
endfm

The next module, CONTINUATION, defines the sort Continuation, which is central to our continuation-based semantics. The constant stop is the empty continuation and denotes a point in which there is no continuation, the end of the program:

fmod CONTINUATION is sort Continuation .  op stop : -> Continuation . endfm

The following module, VALUE, defines the infrastructure for values; the sorts Value and ValueList denote the values and the list of values in the language. As usual, a closure value records the list of formals, the body expression, and an environment. It also stores the number of formals to avoid computing the number of formals in every application. The delayed value records to which location a delayed expression is bound, so that when forced we will be able to update that location with the computed value:

fmod VALUE is including ENVIRONMENT . including CONTINUATION .
  including GENERIC-EXP-SYNTAX .
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  --- values of the language
  op noV : -> Value .  op [_] : ValueList -> Value .
  op int : Int -> Value .  op bool : Bool -> Value .
  op vector (_,_) : Location Nat -> Value .
  op frozen (_) : Exp -> Value .
  op closure : Nat NameList Exp Env -> Value .
  op delayed(_,_,_,_) : Exp Env Location Value -> Value .
  op kon(_) : Continuation -> Value .
endfm

A store is a table mapping locations to values, as the following module, called STORE, shows. Similarly to environments, a store cell has the form [L,V] for any location L and value V, and ___ is the operation that concatenates two stores. The update operation __[_<-_] on stores associates a location list to a value list element-wise.

fmod STORE is protecting LOCATION . including VALUE .
  including ENVIRONMENT . including GENERIC-EXP-SYNTAX .
  sort Store .  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op ____ : Store Store -> Store [assoc comm id: noStore] .
  op [_,<-_] : Store LocationList ValueList -> Store .
... endfm

We finally define the module STATE, which puts together all the infrastructure defined above. A program state is a term having sort MState. It is comprised of a collection of state attributes concatenated with the operator __. The attribute m denotes the store of the program, which will be updated as the program runs. The attribute o contains a list of values that can be output when the program finishes. The attribute d encloses the distinguished top-level environment updated by define expressions. The attribute n is the index of the next free location available in the store. Finally, the attribute k denotes the continuation of the program:
5.2 The Continuation Semantics

The operational semantics of a language is comprised of a set of axioms and inference rules [27,12]. There are essentially two techniques for defining operational semantics: structural (a.k.a. small-step) due to Plotkin [23], and natural (a.k.a. big-step) due to Kahn [14]. Natural semantics is well known for its conciseness, since the transition relation can make big steps of evaluation, but is also known to be deficient in the support of concurrency, since it cannot capture intermediate interleavings of a program. Sequential languages are commonly defined using natural semantics [20]. Therefore, we show a simple informal translation of an inference rule given in big-step style to our semantics.

We take the IMP programming language defined in [27] as an example. In this definition, \( \text{Aexp} \) is the set of arithmetic expressions, \( \text{Loc} \) denotes the set of locations, \( \mathbb{I} \) is the set of integer numbers denoting values\(^2\), and \( \Sigma \) consists of the set of states, that is, functions \( \sigma : \text{Loc} \rightarrow \mathbb{I} \). The transition relation \( \rightarrow_I \subseteq (\text{Aexp} \times \Sigma) \times \mathbb{I} \) is defined inductively on the syntax of arithmetic expressions. The semantics of addition, for instance, can be defined by the inference rule:

\[
\frac{(a_0, \sigma) \rightarrow_I i \quad (a_1, \sigma) \rightarrow_I j}{(a_0 + a_1, \sigma) \rightarrow_I n}, \quad \text{where } n = i + j.
\]

As shown in [21], unconstrained use of structural and natural semantics can lead to unmodular language formalizations. This means that the introduction of a new construct in the language may require modification in rules of other constructs, so one cannot hope that the definitions are definitive. As [21] shows, the undisciplined use of configurations mixing program and instrumental devices as the environment and store can lead to unmodular definitions.

We next informally instantiate the CPS (Continuation Passing Style) semantics [24,18] for the self-contained example of addition. The CPS semantics represents memory states as an (AC) set of attributes whose sort is \( \text{MState} \) as declared in the state infrastructure. This allows one to declare only what is needed, i.e. a projection of the state, in derivations and therefore reduce coupling between definitions and the infrastructure. Furthermore, the CPS semantics is based on continuations. This makes the definition of control-flow extremely simple and provides a systematic way for describing the semantics. The use of continuations is pervasive in programming languages [13,7,4] and for all purposes here we can consider them just as a stack. The dynamic semantics therefore retrieves and pushes continuation items from and to the top of such stack.

In general, the semantics of each language construct can be defined with two equations following a divide-and-conquer methodology. The first equation breaks down an expression into sub-expressions and puts them back on the continuation followed by a mark used to remember what need to be done to complete the evaluation of the original expression. The second equation is in charge of collecting the parts, removing the mark, and computing the value denoting the evaluation of the original expression.

In CPS [18], addition of two expressions \( a_0 \) and \( a_1 \) can be defined as follows:

\[
\vdash k([\text{plus}(a_0, a_1) \circ \text{Env}] \rightarrow K) = k((a_0, a_1 \circ \text{Env}) \rightarrow \{+\} \rightarrow K)
\]

\[
\vdash k([\text{int}(i), \text{int}(j)] \rightarrow \{+\} \rightarrow K) = k(\text{int}(i+j) \rightarrow K)
\]

Note that many of these terms are yet to be defined. However, this simple example gives an evidence that the transition relation \( \rightarrow_I \) can be implemented as a term rewrite system. In fact, these axioms are implemented

\[\text{We changed the set of values to be integers to make the examples uniform with our infrastructure.}\]
as unconditional equations. Note also that we do not need to mention all the attributes in the state. We just specify what we need in order for a derivation step to take place. Here, for instance, we do not mention the store. The variable $K$ stands for the continuation that should be followed after evaluating $\text{plus}(a_0, a_1)$ at environment $\text{Env}$. Note that after evaluating the addition in the domain of integers, the resulting value will be available on the top of the continuation so another “computation” can proceed independently. The following is a non-exhaustive list of features of the CPS semantics:

- it uses an AC list of state attributes;
- it is defined in a continuation passing style;
- expressions carry their evaluation environment.

The first and second design decisions aim at increasing modularity. The second is also central to the definition of control sensitive features, and the third avoids the tedious task of recovering the environment when leaving blocks.

As we illustrate in the next section, the run of a program $E$ is a sequence of the form $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$, where $\rightsquigarrow$ stands for term rewrites, $s_i$ (for $0 \leq i \leq n$) has sort $\text{MState}$, $s_0 = (k([E @ \text{noEnv}] \rightarrow \text{stop}), n(0), d(\text{noEnv}), m(\text{noStore}), o(\text{noVal}))$ is the initial state, and $s_n$ is a state containing the continuation $k(V \rightarrow \text{stop})$. The term $[E @ \text{noEnv}] \rightarrow \text{stop}$ occurring in $s_0$ denotes the current (initial) continuation. It “says” that the expression $E$ waits to be evaluated in the environment $\text{noEnv}$, and nothing else remains to be evaluated. One can understand the equational theory we define as the definition of an abstract state machine implementing the natural semantics of SCHEME.

Note about concurrency

Our CPS definitional methodology informally presented above is capable of translating big-step declarations without compromising concurrent behavior. This perhaps surprising claim [21, page 216] relies on the fact that rewrite rules are applied modulo equations in a rewriting engine like MAUDE. Operationally, equations are applied first, exhaustively in order to “cannonicize” the term/ state to rewrite, then one rewrite rule is applied, then equations again, etc. In order to preserve concurrent semantics, the right notion of thread/process interference needs to be captured using rewrite rules.

One needs to identify which operations affect concurrent behavior. For non-concurrent operations, the “state (term) graph” will commute and therefore the intermediate states do not need to be observable. This technique essentially corresponds to a partial-order reduction [11] on the language semantics. In the case of a language like JAVA [8], only three operations are treated as concurrent: read and write of shared variables, and acquisition of locks. Instead of equations, the semantics of these three operations are defined as rewrite rules in MAUDE using $\text{rl}$ instead of $\text{eq}$. Except for these three cases, every other operation is defined equationally. The small number of rewrite rules improves considerably the performance of state search, as JAVAFAAN demonstrates [8].

Of course, one still needs to increment the infrastructure to support concurrency. Instead of keeping only one continuation in the memory state, one needs to create as many continuations as threads spawned and associate to each thread the set of locks that it holds.

Even though our methodology works well for concurrent language definitions, in this paper we only show it at work on a sequential language definition. However, we believe that SCHEME, and specially its call/cc feature, is complex enough to reflect the strength of our CPS definitional methodology.

5.3 Maude Functional Modules

Due to space limitations we describe only one important subset of modules in the definition of SCHEME. The complete definition is available at [1].

We first present the module GENERIC-EXP-K-SEMANTICS. The continuation $[\_] \rightarrow \_.$ is used throughout the semantics to represent expressions at a given environment. Note that the evaluation of an integer literal is the corresponding integer value. The evaluation of a name, however, needs the store in order to retrieve the associated value. Note that only the continuation, top-level environment, and store attributes “$k, d, m$” are used in that definition. The top-level environment is searched if the name can not be reached from the current environment. The attribute $\text{wise}$ is essentially giving precedence to the previously defined equation.
The continuation \([\_\_]_\to\_\_\) is used to define the evaluation of an expression list into a value list. Finally, a store update occurs when a list of values appears in front of a list of locations on the top of the continuation:

\[
\text{op } [\_\_]_\to\_\_ : \text{ExpList Env Continuation } \to \text{Continuation}.
\]

As usual, in the semantics of \textit{if-then-else} we first evaluate the boolean expression and, depending on the resulted value, we evaluate the \textit{then} expression or the \textit{else}. Note that we rely on the definition of the \textit{Maude} \textit{if-then-else} in the last equation:

\[
\text{fmod IF-K-SEMANTICS is including IF-SYNTAX.}
\]

The following module, \textit{BINDING-K-SEMANTICS}, is needed for the definition of binding expressions. The operator \(a\) has the sole purpose of separating name and expression in a binding list and also of indicating the number of bindings in that list:

\[
\text{fmod BINDING-K-SEMANTICS is including BINDING-SYNTAX.}
\]

The construct \textit{letrec}, defined in the next module, first creates a new environment containing all the names declared, and then evaluate the expressions in the binding list. Finally, the binding is performed and the body evaluated in this new environment. The attribute \(=\) in the conditional performs a pattern-matching when the term on the right cannot be further reduced. We could remove this conditional equation by passing the term \(a\) to the continuation. We decided to keep it for the sake of readability:
The module PROC-K-SEMANTICS below defines the semantics of procedures. The operator function is a convenience to avoid computing the number of formals for every call of the same function. Note that the equation for this operator performs a source-to-source transformation on the program (term). The operation length, defined in the syntactic module NAME-LIST, computes the size of a name list. The function abstraction evaluates to a closure, which is a value of the language (it can be passed around or stored) that also saves the environment (recall that SCHEME is statically-scoped). Function application is defined with two equations. The first evaluates the expressions standing for the closure and actual parameters. The second binds the actual parameters to their formals in the environment saved in the closure. Then, the body is evaluated in the resulting context:

The next module illustrates the semantics of the constructs quote and unquote. The expression quote casts a program fragment to data. It means that the enclosed expression will not be evaluated when quote is. A frozen value includes solely the string of text for a program. The expression unquote takes a frozen value and evaluates it in the current context (SCHEME provides other constructs to facilitate the composition of program and data -- see quasiquotation in [15] -- in lists, but we do not discuss these here):

SCHEME’s call/cc is perhaps the most difficult to understand feature of this language. Despite its complexity it can be defined very concisely in our semantics. This happens mainly because we encode explicitly the continuation in our specification. Note that we introduce a new value, kon(λ), which encloses a continuation. Thus, continuations are regarded first-class citizens; it can be passed around and stored. The expression callcc(λ) is a special kind of function call. When this expression is evaluated, it triggers a call to the function denoted by its parameter, passing the current continuation as a value to this function. Further, this continuation can be "called" like any other function, as the last equation shows. When this happens,
the stored continuation replaces the current and the actual parameters are placed in the top of the new
continuation:

```plaintext
fmod CALLCC-K-SEMANTICS is including CALLCC-SYNTAX . including AEXP-K-SEMANTICS .
    including GENERIC-EXP-K-SEMANTICS . including PROC-K-SEMANTICS . var E : Exp .
    var Xl : NameList . var X : Name . vars K K' : Continuation .
    vars Env Env' : Env . var Vl : ValueList . var # : Nat .
    op kon(_) : Continuation -> Value .
    op [callcc] ->_ : Continuation -> Continuation .
    eq k([(callcc E) @ Env] -> K) = k([E @ Env] -> [callcc] -> K) .
    eq k((closure(#,Xl,E,Env)) -> [callcc] -> K) = k(kon(K) -> [() @ noEnv | closure(#,Xl,E,Env)] -> fn -> K) .
    eq k((kon(K),Vl) -> fn -> K') = k(Vl -> K) .
endfm
```

The module VAR-ASSIGNMENT-K-SEMANTICS below is self-explanatory:

```plaintext
fmod VAR-ASSIGNMENT-K-SEMANTICS is including VAR-ASSIGNMENT-SYNTAX .
    including GENERIC-EXP-K-SEMANTICS .
    var X : Name . var E : Exp . vars Env Env' : Env . var M : Store .
    var K : Continuation . var L : Location . var V : Value .
    op _=>_ : Location Continuation -> Continuation .
    eq k([(set X = E) @ ([X,L] Env)] -> K) = k([E @ ([X,L] Env)] -> L => int(1) -> K) .
    eq k([(set X = E) @ (Env)] -> K) , d([X,L] Env') = k([E @ (Env)] -> L => int(1) -> K) , d([X,L] Env') .
    eq k(V -> L => K) , m(M) = k(K) , m(M[L <- V]) .
endfm
```

The next module, BLOCK-K-SEMANTICS, defines sequential composition of expressions:

```plaintext
fmod BLOCK-K-SEMANTICS is including BLOCK-SYNTAX .
    including GENERIC-EXP-K-SEMANTICS .
    op ignore -> _ : Continuation -> Continuation . var E : Exp . var El; : ExpList; .
    var Env : Env . var K : Continuation . var V : Value .
    eq k([(E) @ Env] -> K) = k([E @ Env] -> K) .
    eq k([(E ; El;) @ Env] -> K) = k([E @ Env] -> ignore -> ([El;] @ Env) -> K)) .
    eq k(V -> ignore -> K) = k(K) .
    eq k(ignore -> K) = k(K) .
endfm
```

The module DELAYED-EXPRESSION-K-SEMANTICS implements the "promises" data-type. A promise results from
the evaluation of a delay(\_) expression. In contrast to quote, this expression stores the current environment
which will be used to evaluate the stored expression in the future. This data-type can be used to model
call-by-need. The construct force(\_) is used to reclaim the promise. Note that if such value is stored at some
location, then the result of the evaluation will be memorized at the same location. This avoids the expression
to be evaluated several times:

```plaintext
fmod DELAYED-EXPRESSION-K-SEMANTICS is including PL-SYNTAX .
    including DELAYED-EXPRESSION . including GENERIC-EXP-K-SEMANTICS .
    var E : Exp . var Env : Env . var V : Value .
    var K : Continuation . var L : Location . var M : Store .
    op [force] -> _ : Continuation -> Continuation .
    op [memo_] -> _ : Value Continuation -> Continuation .
    eq k([delay(E) @ Env] -> K) = k(delayed(E, Env , noL , noV) -> K) .
    eq k([force] @ Env] -> K) = k ([E @ Env] -> [force] -> K) .
    eq k(delayed(E,Env,noL,noV) -> [force] -> K) = k([E @ Env] -> K) .
    eq k(delayed(E,Env,L,noV) -> [force] -> K) = k([delayed(E,Env,L,noV) -> [force] -> K) .
    eq k(delayed(E,Env,L,noV) -> [force] -> K) = k([delayed(E,Env,L,noV) -> K] .
    --- if the promise is evaluated
    eq k(delayed(E,Env,L,V) -> [force] -> K) = k (V -> K) [owise] .
    eq k(V -> [memo(delayed(E,Env,L,noV)))) -> K) , m(M)
        = k(V -> K) , m(M[L <- delayed(E,Env,L,V)]) .
endfm
```
The semantics of SchemeM is obtained by including the semantics of the features discussed above (and others). We define an operator eval in terms of a set of state attributes, denoting the “initial state”. In this set we add all the state attributes required to perform an evaluation, and include the expression (the program) on the top of the continuation, with an empty environment. The evaluation terminates when the stop continuation is reached:

fmod PL-K-SEMANTICS is protecting PL-SYNTAX.
  including AEXP-K-SEMANTICS . including BEXP-K-SEMANTICS .
  including VECTOR-K-SEMANTICS . including LIST-K-SEMANTICS .
  including IF-K-SEMANTICS . including LET-STAR-SEMANTICS .
  including LET-K-SEMANTICS . including PROC-K-SEMANTICS .
  including LITERAL-K-SEMANTICS . including CALLCC-K-SEMANTICS .
  including LETREC-K-SEMANTICS .
  including VAR-ASSIGNMENT-K-SEMANTICS .
  including BLOC-K-SEMANTICS . including LOOP-K-SEMANTICS .
  including CASE-K-SEMANTICS . including IO-K-SEMANTICS .
  including DEFINE-K-SEMANTICS .
  including DELAYED- EXPRESSION-K-SEMANTICS .
  op eval : Exp -> Value . op [] : MState -> Value .
  var V : Value . var S : MState . var I : Int .
  var K : Continuation . var V1 : ValueList . var X : Name .
  eq eval(E) = [k([E @ noEnv] -> stop)), n(1), m(noStore), o(noVal), d(noEnv)] .
  eq [k(V -> stop), S] = V .
  eq [k(stop), S] = noVal .
endfm

6 Performance Results

Figure 1 shows a factorial program implemented with callcc. The other programs mentioned in Table 1 appear in Appendix B. We assume they implement reasonably standard algorithms and omit them in this paper.

```scheme
let g =
  lambda ('v1,'v2) ->
    let r = lambda ('kon) -> list ('kon , 'v1 , 'v2)
    in callcc(r)
in let 'fact-callcc =
  lambda (v) ->
    let 'triple = g(1,v)
    in let 'first = car('triple),
        'second=car(cdr('triple)),
        'third=car(cdr(cdr('triple))
    in {
      if eq('third,0)
        then 'second
        else 'first (  
            list('first , times ('second , 'third ), minus('third , 1 ) )
        )
    }
in 'fact-callcc(25000)
```

Fig. 1. Factorial via callcc
Table 1 shows the running times and the number of rewrites (inside parenthesis) of various programs in a benchmark comparing our implementation of SCHEME with that of Dr.Scheme, a well-known C implementation of SCHEME. We used a two (2.4GHz and 512KB cache) processor machine with 4GB of memory running Fedora Core release 1. The SCHEME programs are listed in Appendix B. Note that the running times of the first four entries are close to that of Dr. Scheme. We believe that the highly efficient implementation of lists in Dr. Scheme might be the reason for the difference in the last 2 entries. We included in this table entries to the recursive and iterative factorial of 300. These were also used in [6], where they achieved results of 0.45 seconds and 23,122 rewrites, and 0.52 seconds and 33,511 rewrites for the recursive and iterative implementations, respectively. Note, however, that [6] implements a different language using a different methodology.

<table>
<thead>
<tr>
<th>Program</th>
<th>SCHEME</th>
<th>Dr.Scheme mzScheme mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact-recursive(300)</td>
<td>0.040</td>
<td>(12,319)</td>
</tr>
<tr>
<td>fact-recursive(25000)</td>
<td>9.170</td>
<td>(1,025,019)</td>
</tr>
<tr>
<td>fact-iterative(300)</td>
<td>0.030</td>
<td>(11,749)</td>
</tr>
<tr>
<td>fact-iterative(25000)</td>
<td>7.440</td>
<td>(975,049)</td>
</tr>
<tr>
<td>fact-callcc(25000)</td>
<td>15.490</td>
<td>(2,100,197)</td>
</tr>
<tr>
<td>insert-sort(400)</td>
<td>43.840</td>
<td>(4,269,102)</td>
</tr>
<tr>
<td>permutations-up-to-n(8)</td>
<td>112.770</td>
<td>(12,776,619)</td>
</tr>
</tbody>
</table>

Table 1. Running times in seconds for a benchmark

It is expected that compilation of equational specifications can lead to significant speed-ups. We hope that in the foreseeable future, our approach can not only serve to prototyping and analyzing programs, but also as a way to deliver very efficient language interpreters. The source code and benchmarks of SCHEME can be downloaded at [1].

7 Conclusions

This paper presents the semantics of the language SCHEME (R5RS) as an equational theory in the MAUDE rewriting system. We claim that the definition is highly modular and thus can be easily extended with new constructs. Similar semantics for JAVA, JVM, OCAML, ML, and HASKEL [2,8] have already been or are currently being developed in the Formal Systems Laboratory at the University of Illinois at Urbana-Champaign.

We strongly believe that these efforts might also set the ground for the development of software analysis tools. MAUDE already provides the user with a breadth-first search and LTL model-checker for theories developed in it. Recently, our group implemented dimensional analysis of C programs [25]. In [24], students defined the Hindley-Milner $W$ [19] type inference algorithm in a two-week assignment. We currently investigate the use of equational logics in the development of static analyzers. In particular, the definition of symbolic execution tools and abstract interpreters [9, ?] using some of the techniques presented in this paper.

References


Appendix A. Syntax of SchemeM

We present the grammar of a language similar to Scheme, here called SchemeM. We made modifications to the syntax to make parsing easier. Each of the following MAude functional modules describe a different construction of the language. Note that such syntactic modules have no equations. These modules are only concerned with parsing of terms.

```maude
fmod NAME is including QID .
  sort Name .
  subsort Qid < Name .
  ops a b c d e f g h i j k l m n o
       p q r s t v w x y z w : -> Name .
endfm
```

The subsort construct of MAude used in this module states that Qid has kind Name. In other words, a Qid can be provided wherever a Name is expected. It means that the quoted text ‘aString is also a name in our specification.

```maude
fmod NAME-LIST is including NAME .
  sort NameList . subsort Name < NameList .
  op '(' : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: ()] .
  op length_ : NameList -> Nat . eq length() = 0 .
  eq length(X,Xl) = 1 + length(Xl) .
endfm
```

The attribute assoc states that the operator , is associative. This is necessary to avoid parentheses in declaration of name lists. In addition, the empty list () is the identity for name lists. So, for any name list nl, the lists (),nl and nl are equivalent.

Another useful attribute (not used here) is comm. It states that a mixfix operator is commutative. These three (assoc, comm, and id:) attributes are central to rewriting since equations are applied modulo ACI (associativity, commutativity, and identity). That is, if a term can be matched modulo ACI to the left-hand-side of an equation (see section 3) it can be rewritten to the term denoted by the right-hand-side. The attribute prec defines precedence of terms. It is used to solve ambiguities of the parser.

```maude
fmod GENERIC-EXP-SYNTAX is
  including NAME .
  including INT .
  sort Exp .
  subsorts Int Name < Exp .
endfm
```

```maude
fmod EXP-LIST is
  including GENERIC-EXP-SYNTAX .
  including NAME-LIST .
  sort ExpList .
  subsorts Exp NameList < ExpList .
  op noExp : -> Exp .
endfm
```

The attribute ditto indicates that the set of attributes for this operator are the same as those defined for the subsorts of ExpList. In this case, NameList. If subsorts have inconsistent attributes a message is given by MAude.

```maude
fmod ARITHMETIC-EXP-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  ops plus minus times div mod : Exp Exp -> Exp .
endfm
```
fmod BOOLEAN-EXP-SYNTAX is
  including GENERIC-EXP-SYNTAX.
  ops true false : -> Exp.
  ops eq leq geq and or : Exp Exp -> Exp.
  op not : Exp -> Exp.
endfm

fmod LIST-SYNTAX is including EXP-LIST.
  op <_> : ExpList -> Exp.
  op list : ExpList -> Exp.
  ops car cdr : Exp -> Exp.
  op cons : Exp Exp -> Exp.
  op emptyList : -> Exp.
  ops map append : Exp Exp Exp -> Exp.
  ops reverse length : Exp -> Exp.
  ops set-car set-cdr : Name Exp -> Exp.
endfm

fmod VECTOR-SYNTAX is including EXP-LIST.
  op vector_ : ExpList -> Exp.
  op make-vector : Nat -> Exp.
  op make-init-vector : Nat Exp -> Exp.
  op vector-length : Exp -> Exp.
  op vector-ref : Exp Exp -> Exp.
endfm

fmod LITERAL-SYNTAX is including GENERIC-EXP-SYNTAX.
  op quote : Exp -> Exp.
  op unquote : Exp -> Exp.
endfm

fmod DELAYED-EXPRESSION is including GENERIC-EXP-SYNTAX.
  op delay : Exp -> Exp.
  op force : Exp -> Exp.
endfm

fmod BLOCK-SYNTAX is including GENERIC-EXP-SYNTAX.
  sort ExpList;.
  subsort Exp < ExpList;.
  op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100].
  op { } : ExpList; -> Exp.
endfm

fmod IF-SYNTAX is including GENERIC-EXP-SYNTAX.
  op if_then_else_ : Exp Exp Exp -> Exp.
endfm

fmod CASE-SYNTAX is including EXP-LIST.
  sorts CaseMatching CaseMatchingSeq.
  subsort CaseMatching < CaseMatchingSeq.
  op [_] _ : ExpList Exp -> CaseMatching [prec 100].
  op emptyCaseMatching : -> CaseMatching.
  op _ _ : CaseMatching CaseMatchingSeq -> CaseMatchingSeq [prec 101 id: emptyCaseMatching].
  op case_in_ : Exp CaseMatchingSeq -> Exp [prec 102].
  op case_in_ , else _ : Exp CaseMatchingSeq Exp -> Exp [prec 102].
endfm
We finally define the module \texttt{PL-SYNTAX} (not shown here) including all these syntactic modules, which corresponds to the entire \texttt{SCHEMEm} syntax.

Figure 2 shows an example of equivalent programs in \texttt{SCHEMEm} and \texttt{SCHEME}.
let x = 5 in
let rec
    f = lambda (y) -> c(x, y),
    c = lambda (a , b) ->
        plus( times(a,b) , a)
    in f(plus(x , 3))

Fig. 2. Equivalent procedures written in SCHEME M and SCHEME

Appendix B. Benchmark

{ define x = 10000 ;
    let rec fact-recursive =
        lambda (n) -> if eq(n , 1)
            then 1
            else times(n ,
                fact-recursive(minus(n , 1)))
    in (f x) }

Fig. 3. recursive factorial

{ define x = 50000 ;
    let fact-iterative = lambda (n) ->
        let p = 1 , i = 2 in {
            while leq(i , n) {
                set p = times( p , i ) ;
                set i = plus( i , 1 )
            }
        } ;
    in fact-iterative(x)
}

Fig. 4. iterative factorial
letrec

    insert = lambda (x,l) ->
        if null?(l)
            then list(x)
        else if leq(x, car(l))
            then cons(x,l)
        else cons(car(l), insert(x,cdr(l)))
    , insert-sort = lambda (l) ->
        if null?(l)
            then emptyList
        else insert(car(l), insert-sort(cdr(l)))
    , genlist = lambda (n) ->
        if eq(n, 0)
            then emptyList
        else cons(n, genlist(minus(n,1)))
    in insert-sort(genlist(200))

Fig. 5. insertion sort

letrec

    permutations-up-to-n = lambda (n) ->
        if eq(n, 0)
            then list(emptyList)
        else insert(n, permutations-up-to-n(minus(n,1)))
    , insert = lambda (n,l) ->
        if null?(l) then emptyList
        else app(interleave(n,car(l)), insert(n,cdr(l)))
    , interleave = lambda (n,l) ->
        if null?(l) then list(n)
        else cons(cons(n,l), mycons(car(l),
            interleave(n,cdr(l))))
    , mycons = lambda (x,l) ->
        if null?(l)
            then emptyList
        else cons(cons(x,car(l)), mycons(x,cdr(l)))
    , app = lambda (u,v) ->
        if null?(u) then v else cons(car(u), app(cdr(u), v))
    in permutations-up-to-n(7)

Fig. 6. list of \(n!\) permutations over the first \(n\) natural numbers

19