# A Formal Rewriting Logic Semantic Definition of Scheme ⋆

Patrick Meredith, Mark Hills, and Grigore Roşu

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

**Abstract.** This paper presents a formal definition of Scheme (based on the informal definition given in the R$^5$RS report [12]). The definition is purely equational, so it can be regarded as an *algebraic denotational specification* with an initial model/algebra semantics of Scheme. Moreover, it is *executable*, in the sense that equations can be oriented from left-to-right into rewrite rules and thus giving an operational semantics of Scheme as well; this way, an interpreter for Scheme is obtained for free by just executing the presented Scheme definition on term rewrite engines. Maude is used in this paper, but other equational engines could have been used as well. The definition in this paper is the most complete formal definition of Scheme that we are aware of and can play two important roles: as a formal definition of Scheme complementary to the informal one in the R$^5$RS report, and as a platform for experimentation with variants and extensions of Scheme, for example concurrency. This work is part of the *rewriting logic semantics* project, whose broad scope is to formally define languages and language features in rewriting logic, and then use the generic support provided by rewriting logic to obtain not only interpreters, but also formal analysis tools for the defined languages.

**keywords:** Semantics, rewriting, Scheme.

## 1 Introduction

Scheme is a general purpose programming language with a unified handling of data and code. It also has a powerful macro system, using pattern matching, to express syntax transformations. The Revised$^5$ Report on the Algorithmic Language Scheme (R$^5$RS [12]) gives a thorough, but informal description of the language, as well as a partial denotational semantics. The denotational semantics in [12] is missing definitions of important language features, such as definitions of `eval` and `dynamic-wind`, it does not define the "top level" used throughout the informal specification, and, most importantly, it is not executable. Executability of a language definition gives one confidence in the appropriateness of the definition. Indeed, one can execute hundreds of programs exercising various language

⋆ Supported by NSF CCF-0448501 and NSF CNS-0509321.

features or combinations of features, and thus find and fix errors in the definition. Many subtle errors were detected and fixed in our subsequent definition due to its executablity.

More recent attempts have been made at giving formal, operational/executable semantics to fragments of Scheme [15, 5]. Unfortunately, the partial definition in [5] does not use a proper representation for vectors and lists, so it cannot be extended to the complete Scheme, and neither [15] nor [5] gives definitions for `quasiquote` or macros. Furthermore, neither uses a unified representation of data and code, which is one of the crucial defining aspects of Scheme. These approaches, their limitations and comparisons with our current definition are further discussed in Section 4.

In this paper we introduce a novel formal executable definition of the Scheme programming language, based on $R^5RS$. Our definition uses a proper representation for lists and vectors, a unified representation of code and data, and defines `quasiquote` and a large portion of `define-syntax` macros. This definition uses the K definitional technique [23] within rewriting logic [17], so we refer to it as K-Scheme. K is a language definitional framework consisting of the K-technique, based on a first-order representation of computations as lists or stacks of "computational tasks", and of the K-notation, a domain-specific notation within rewriting logic that eases understanding and defining programming languages; we do not use the K notation in this paper. Rewriting logic is a unified logic for concurrency that extends equational logic with transitions; we use only the equational fragment of rewriting logic in this paper.

We used the equational system Maude [3] to implement our equational definition of K-Scheme. Currently, K-Scheme consists of 772 equations, 192 of them for `define-syntax` macros and 575 for the core of the language (and a few built-in procedures). We define 60 features of Scheme, using 310 auxiliary operators and 2152 lines of Maude code; 374 lines of code, however, define aspects of the K framework also common to other language definitions, and simple helping operations. Also, we note that many features of Scheme for which we give Maude code definitions could be written as Scheme macros (e.g., `let` and `cond`), but we did not follow that approach.

The complete Maude definition of K-Scheme can be found on K-Scheme's webpage at [16], together with a web interface allowing one to "execute" programs directly within K-Scheme's definition, using Maude's capability to execute equational specifications. The main limitations of K-Scheme at this point are an incomplete standard library, lack of internal `define`'s, and the support of only integers among the numeric types. These, as well as other implementation-specific features of Scheme, can be added modularly (i.e., without having to modify the definitions of the existing features) and will be added eventually. Nevertheless, this is the most complete formal definition of Scheme of which we are aware. In particular we believe we are first to give formal definitions to the opera-

tions of `quasiquote`, `unquote`, `unquote-splicing`, and a partial definition of `define-syntax`.

***On Rewriting Logic Semantics and K.*** This paper is part of the rewriting logic semantics (RLS) project (see [20, 18] and the references there). The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on *rewriting logic* [17]. It has been shown in [25] that conventional definitional styles, such as big-step [11] and small-step SOS [22], MSOS [21], reduction semantics with evaluation contexts [29], the chemical abstract machine [1], and continuation-based semantics, can all be faithfully captured, in the sense of intended computational granularity, as rewrite logic theories. Therefore, rewriting logic can be indeed used as an ecumenical framework for language definition using any of the above-mentioned styles, inheriting all their advantages and disadvantages.

K [23] is an attempt to optimize the use of rewriting logic for language definitions without obeying any of the styles above; it is, though, closest in spirit to continuation-based semantics, in that it maintains the current computation as a special structure that can be manipulated like any other data-type, in particular altered. The K technique uses a subset of rewriting logic and can be easily supported by other frameworks, for example by functional programming systems; however, in that case one would use K for the sole purpose of implementing interpreters.

Since our definition of Scheme in this paper is purely equational, we indirectly show that equational logic and algebraic specification are powerful enough to formally define real languages, if used properly, such as in the K framework. We invalidate once again, experimentally, the unfortunately common belief that equational logic and standard initial algebra semantics are insufficient to deal with complex languages. In fact, K-Scheme is an initial algebraic model for the Scheme language, a non-trivial language. Moreover, the equations we use to define K-Scheme can be executed as rewrite rules by equational engines with support for rewriting, such as Maude, giving us an "interpreter" essentially for free.

## 2 Rewriting Logic Semantics

This section provides a brief introduction to term rewriting, rewriting logic, and the use of rewriting logic in defining the semantics of programming languages. Term rewriting is a standard computational model supported by many systems; rewriting logic [17, 14] organizes term rewriting modulo equations as a complete logic and serves as a foundation for programming language semantics [18–20]. Continuation-based rewriting logic semantics, the form of rewriting logic semantics adopted in this paper, provides explicit representations of control context which can be used in the definitions of language features that manipulate this context, such as continuations, exceptions, or jumps.

### 2.1 Term Rewriting

Term rewriting is a method of computation that works by progressively changing (rewriting) a term. This rewriting process is defined by a number of rules – potentially containing variables – which are each of the form: $l \rightarrow r$. One step of rewriting is performed by first finding a rule that matches either the entire term or a sub-term. This is done by finding a substitution, $\theta$, from variables to terms such that the left-hand side of the rule, $l$, matches part or all of the current term when the variables in $l$ are replaced according to the substitution. The matched sub-term is then replaced by the result of applying the substitution to the right-hand side of the rule, $r$. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a sub-term, rule, and substitution such that $\theta(l)$ matches the sub-term. When no matching sub-terms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, may not terminate.

There exist a plethora of term rewriting engines, including ASF [27], Elan [2], Maude [3], OBJ [8], Stratego [28], Tom [13], and others. Rewriting is also a fundamental part of existing languages and theorem provers. Term rewriting is inherently parallel, since non-overlapping parts of a term can be rewritten at the same time, and thus fits well with current trends in architecture and systems.

### 2.2 Rewriting Logic

Rewriting logic is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the $\lambda$ calculus where $\lambda$ terms can be grouped into equivalence classes based on relations such as $\alpha$ and $\beta$ equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency, where different orders of evaluation could lead to non-equivalent results, such as in the case of data races. The distinction between rules and equations is crucial for formal analysis, since terms which are equal according to equational deduction can all be collapsed into the same analysis state. Rewriting logic is connected to term rewriting in that all the equations and rules of rewriting logic, of the form $l = r$ and $l \Rightarrow r$, respectively, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming both into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term and "executing" it.

### 2.3 Maude: A Rewriting Logic System

In this paper we discuss a rewrite logic definition of Scheme using Maude [3], a high-performance rewriting logic system. In Maude, equations are defined as `eq`

`l = r`, while rules are defined as `rl l => r` (the `=>` symbolizing a one-way transition, versus an equality). Conditions may be added to both equations and rules, with conditional equations represented as `ceq l = r if c` and conditional rules represented as `crl l => r if c`. Terms, such as `l`, `r`, and `c` above, are formed from operations, defined using the keyword `op`, and from variables, declared using the keyword `var`; by convention, operator names start with lowercase letters or symbols, while variable names start with uppercase letters. Equations and rules can be used directly to execute a program based on a rewriting logic definition. In this paper, since K-Scheme does not include concurrency features, we only make use of equations, giving us a standard algebraic specification with initial algebra semantics.

Maude provides several capabilities beyond standard equations and rules which make it useful for defining languages and performing formal analysis of programs. Maude allows commutative and associative operations with identity elements, allowing straight-forward definitions of language features which make heavy use of sets and lists, such as sets of program state information and lists of computational tasks. Maude also provides built-in support (not explored in this paper) for model checking and breadth-first state space exploration, using the rules defined in the semantics to indicate competing tasks (memory accesses, lock acquisition, etc) which can split the state space.

### 2.4 K: A Continuation-based Rewriting Logic Semantics

K [23] is a rewriting logic semantics framework consisting of a technique and a specialized notation, to define programming languages as rewriting logic theories. In this paper, we use the K technique to define Scheme in Maude; the reader interested in the K notation, as well as in further details regarding the K technique, is referred to [23]. By K, we understand the K definitional technique within rewriting logic.

In K, the current program is represented as a potentially nested "soup", (or multi-set), of terms representing the current computation, memory, global definitions, etc. Information stored in the state can be nested, allowing logically related information to be grouped and manipulated as a whole. The most important piece of information is the `Continuation`, wrapped by the operator `k`, which is a first-order representation of the current computation, made up of a list of computational tasks separated by `->`. The continuation can be seen as a stack, with the current computational task at the left and the remainder (continuation) of the computation to the right. This stack, along with other state components, can be saved and restored later, allowing complex control structures to be defined. For example, if in a certain definitional context where the remaining computation is represented by continuation `K` one wants to schedule for processing/evaluating expression `E`, all one needs to do is replace the current continuation in the state configuration by `E -> K`. After `E` evaluates to value `V` the continuation will be `V -> K`.

Lists, used frequently in K, and Maude definitions in general, are defined in Maude as associative operations with identity elements. An example is `ValueList`:

```
sort ValueList .
subsort Value < ValueList .
op nill : -> ValueList .
op _,_ : ValueList ValueList -> ValueList
         [assoc id: nill] .
```

Here, `ValueList` can be seen as a new "type", or sort; `Value` is declared to be a subsort, meaning that a `Value` can be treated as a (trivial, one element) `ValueList`. `nill` is declared as an operation with no arguments, also called a constant, of result sort `ValueList`, and is made the identity of the list formation operation, `_,_`, which allows us to assume that the list always has a tail (since we can always add `nill` to the end of the list). List formation is associative, allowing us to arbitrarily group elements in the list, but is not commutative, since order is important.

Multisets are defined similarly in Maude, but are also commutative. An example is `Env`, which represents environments:

```
sort Env .
op empty : -> Env .
op [_,_] : Name Location -> Env .
op __ : Env Env -> Env [assoc comm id: empty] .
op _[_] : Env Name -> Location .
op _[_<-_] : Env Name Location -> Env .
eq ([X,L] Env)[X] = L .
eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
eq Env[X <- L] = (Env [X,L]) .
```

Here, again, a new sort, `Env`, is defined. An environment is either `empty` or is a pair of `Name` and `Location`. We can also form an environment by putting it next to another environment, forming an environment set using the `__` operation; this operation is associative and commutative, with `empty` as the identity. The final two operations allow environment lookup (`_[_]`) and modification (`_[_<-_]`). The definition of these two operations is shown in the three equations. The first defines lookup: when looking up name `X`, if there is a pair with name `X` and location `L` in the environment (`Env` is the rest – since environments are sets and are defined as commutative, we can always assume that the pair we are interested in is the first pair in the set), return location `L`. The next two equations define environment modification. In the first, the name `X` is already in the environment with location `L`, and we want to change this to location `L'`, so we update the existing pair, leaving the rest unchanged. In the second, `X` was not found in the existing environment, so we just add a new pair with name `X` and location `L`. The first two equations make use of Maude's complex *matching modulo equations*, in this case modulo associativity and commutativity (of operator `__`). Here, and in the rest of the paper, we rely on Maude's order of equation application, since the final equation would also encompass the case in the second equation. One can easily translate our current definition into a Maude definition that is *not* dependent on equation ordering, making use of Maude features such as "otherwise" and conditional equations, but we do not

do this here since these translations tend to make the equations more complex. Methods of automatically translating definitions using these features into pure rewriting logic specifications that maintain the proper algebraic semantics are available for both otherwise [4] and conditional equations [24].

Using lists and sets (combined with the appropriate state infrastructure, defined in Section 3), we can then define more complex equations such as:

```
eq k((V,Vl) -> assignTo(X,Xl) -> K) mem(Mem)
   env([X,L] Env)
 = k(Vl -> assignTo(Xl) -> K) mem(Mem[L <- V])
   env([X,L] Env) .
```

This equation assigns a list of values (computed based on a list of expressions) to a list of names. `V` represents a value at the head of the list of values, with `Vl` representing the tail. Similarly, `X` represents a name at the head of the list of names, with `Xl` as the tail. `assignTo` is a *continuation item*, and is used to represent the action of assigning computed values to a list of names. `K` matches the rest of the computation – i.e., the next computational steps once the assignment is complete. `Mem` represents the current memory, a mapping of locations to values, while `Env` represents an environment, (as shown above) a mapping of names to locations. `L` represents one of these locations.

The equation works as follows: given a name `X`, look up the location at which `X` is stored. This leverages matching modulo associativity and commutativity (set or multi-set matching) twice, first to bring in both the continuation and the environment parts of the "soup", and second to identify the proper name/location pair in the environment. With this location, the term representing the computation can be modified, representing a step of computation. This step will leave the remaining parts of the assignment on the continuation (the remaining values and names), will leave the environment unchanged, but will modify the memory, replacing it with an altered memory where location `L` takes on value `V`. This will use memory update equations like those for `Env` shown above.

## 3 Scheme in Rewriting Logic

In our K definition of Scheme, K-Scheme, we attempted to cover the *entirety* of core Scheme as defined, informally, in R$^5$RS [12]. By "core" we mean those syntactic keywords and procedures not marked as library. We also support select library syntax and procedures, and intend to offer a full standard library in the future. Specifically, our K-Scheme currently includes formal definitions for the following Scheme features: `+`, `-`, `*`, `/` (integer only), `append`, `and`, `apply`, `begin`, `boolean?`, `call-with-current-continuation`, `call-with-values`, `car`, `cdr`, `cadr`, `cddr`, `char?`, `cond`, `cons`, `define`, `delay`, `do`, `define-syntax`, `dynamic-wind`, `eq?`, `equal?`, `eqv?`, `eval`, `expt`, `force`, `if`, `lambda`, `let`, `let*`, `letrec`, `list`, `display`, `make-string`, `make-vector`, `not`, `null?`, `number?`, `or`, `pair?`, `procedure?`, `string-length`, `string-ref`, `string-set!`, `symbol?`, `syntax-rules`, `quote`,

`quasiquote`, `set!`, `set-car!`, `set-cdr!`, `unquote`, `unquote-splicing`, `vector?`, `vector-length`, `vector-ref`, and `vector-set!`.

In terms of core syntactic keywords, we lack proper support for internal defines. All defines act as if they are at the top level. R$^5$RS mentions, however, that internal defines can be rewritten as `letrec`. We support `quasiquote` and the `define-syntax` form of macros, which we consider part of the core language. The macro support is not complete, but many standard examples can be handled by our definition. We are primarily lacking only those core procedures which operate on different types of numbers, since K-Scheme currently only supports integers. The predicate `number?` returns `#t` for any integer. Conversions between different data types are currently missing, but are easy to define. Characters, while defined, are currently missing comparison operators. Input and output are currently limited to the procedure `display`. The complete definition of K-Scheme using the K technique can be found on K-Scheme's webpage at [16].

### 3.1 Syntax

To make Scheme more palatable to Maude, the internal definition of the Scheme syntax used by K-Scheme is slightly different from standard Scheme syntax. We provide an external parser at [16] capable of converting normal Scheme code to K-Scheme code, but currently output from programs uses K-Scheme syntax[1].

Parenthesis are significant to Maude (to resolve precedence conflicts), so we removed them in favor of square brackets. Note that many Scheme interpreters already allow the use of square brackets to denote parentheses. Maude expects strings to be delimited by double quotes, so strings in our syntax are wrapped in curly braces, for example {`"foo"`}. Character constants use the same "`#\`" syntax as Scheme, save that Maude characters are single character strings, so a character in K-Scheme looks like `#\("f")`. The short-cut syntax for `quote`, `quasiquote`, `unquote`, and `unquote-splicing` needed to be changed, because "`'`", "`` ` ``", and "`,`" are all significant to Maude. We therefore use "`$`", "`!`, "`!!`", and "`!@`" respectively. All variable names other than the 26 letters of the alphabet need to be quoted. All examples in this paper are given in normal Scheme syntax, but in some places K-Scheme syntax is used to show output. Also, some of the macro equations use the K-Scheme syntax because they are syntax transformations.

### 3.2 Scheme State Representation

When defining a language using K, one of the important decisions is the structure of the state. By "state", we here mean all the information about a program execution snapshot, including the program itself; in this sense, it is like a "configuration" in SOS [22]. The rewrite rules require this state structure to determine the context of equation application. The major concerns are that all needed

[1] We must stress that programs can be written in normal Scheme syntax due to our external parser at [16]

information be available, and that the state is organized in a logical, extensible manner. Our goal is for additions to the state representation to be possible *without* breaking existing equations in the semantics, when possible, and vice versa.

The state representation for K-Scheme consists of the components: `k`, the continuation; `mem`, the store; `nextLoc`, the next free location in the store; `env`, the local environment; `globalenv`, the global environment; `synmap` the syntax map for macros (see Section 3.10); `output`, the output of the program; and `program`, the stored syntactic representation for the rest of the program not currently in the continuation.

As the heart of computation, `k` is the predominate feature in most of the equations. An effort is maintained to match only the front of the continuation in equations, both for ease of understanding, and efficiency reasons.

The store, `mem`, contains all program values bound to variables, or contained in structures bound to variables. It is a mapping from location (given as the constructor `loc` with a natural number argument, though any sort with a partial ordering and an increment operator could be used in place of natural numbers) to program values. The natural number value of `nextLoc` is incremented after every allocation, ensuring that previous store values are not lost.

The environments, `env` and `globalenv`, map program variables (symbols) to locations. A distinction between local environment and global environment is necessary in the presence of closures. When a closure is formed, only the local environment is saved in the closure, thus changes to the referent of variables mapped in the global environment will be visible. This necessity is explained further in Section 3.6. The separation of store from the variable binding of the environment allows for an easier representation of the complex structures in Scheme (e.g. lists, vectors).

Every time the built-in function `display` is called in a program, its argument is converted to a string representation and appended to the value contained in the `output` state component.

The last component, `program`, contains the syntactic representation of all expressions in the program not currently executing. K-Scheme allows for multiple expressions, which are computed in order, as one would expect from a Scheme program given to an interpreter or compiler in a non-interactive mode. The presence of `call-with-current-continuation` (`call/cc`) necessitates `program`. `call/cc` requires the entire continuation be captured at the point of call, and passed to its argument. If the entire program exists in the continuation, then the entire program would be passed to the argument of `call/cc`; this is not the desired behavior, and can result in unexpected non-termination cases (see Section 3.7).

The following is the equation creating the initial state:

```
eq run(EL) = [k(initialize -> stop) mem(empty)
              program(EL) globalenv(empty) env(empty)
              synmap(empty) nextLoc(1) output(none)] .
```

When the `run` operator is applied to an `ExpressionList EL`, we place the `initialize` operator followed by `stop` on the continuation. `EL` is placed in the `program` attribute. `initialize` will bind the built-in functions and syntactic keywords to their names; it also defines some built-in procedures via actual scheme code. `stop` is a signal to the definition to place the next `Expression` in the `program` attribute onto the continuation, or, if none exists, to end execution. Recall that the "computational tasks" listed in the continuation with the construct "`->` " are processed in order from left to right. We pass empty environments and stores to `env`, `globalenv`, and `mem`, but these will be populated during initialization. `synmap` is also initialized to be empty. This equation showcases well the attributes of K-Scheme's state. Note that equations need only reference attributes of the state significant to their operation.

### 3.3 Lists

The aspect of Scheme that we consider as the most important characteristic of the language case study in this paper is the unified representation for both program and data. All functioning programs are lists. To support the semantics of lists, we use a storage model much like that given in the R$^5$RS report [12].

Internally, all lists are represented as cons cells. Cons cells are pairs of locations, which can be thought of as pointers. To form an actual list, the second location, the `cdr` of the cons cell, points to another cons cell. We chose this representation both because it is the representation suggested by R$^5$RS and because it easily supports desired Scheme functionality. An example is the sharing of `cdr`'s. Two lists may share `cdr`'s, wherein the update to the `cdr` of one list is reflected in the `cdr` of the list sharing that `cdr`:

```
(define x '(1 2 3 4))
(define y '(1 . 2))
(set-cdr! y (cdr x))
y ===> (1 2 3 4)
(set-car! (cdr x) 9)
x ===> (1 9 3 4)
y ===> (1 9 3 4)
```

Because we represent cons cells as pairs of locations, the `cdr`'s of the cons cells representing `x` and `y` in the above example point to the same physical cons cell, and any updates will be reflected in both. The Maude syntax for cons cell is:

```
sort ConsCell .
op {_._} : Location Location -> ConsCell .
```

This structure also allows the built-in Scheme operations on lists to be handled fairly trivially. Finding the `car` of a list, `x`, simply amounts to looking up the value pointed to by the first location in the cons cell representing `x`.

Recall that due to the `program` state attribute we only execute one expression in the continuation at a time. These expressions, however, can be arbitrarily complex. Each complete expression is first converted into this list representation (before execution). Execution is on list structures consisting of cons

```
eq k(apply(fbuiltin(car), cell({L1 . 2})) -> K) mem(Mem)
 = k((Mem[L1]) -> K) mem(Mem) .
eq k(apply(fbuiltin(cdr), cell({L1 . L2})) -> K) mem(Mem)
 = k((Mem[L2]) -> K) mem(Mem) .
eq k(apply(fbuiltin(set-car!),cell({L1 . L2}),V) -> K)
 = k(V -> assignToLoc(L1) -> symbol(unspecified) -> K) .
eq k(apply(fbuiltin(set-cdr!),cell({L1 . L2}),V) -> K)
 = k(V -> assignToLoc(L2) -> symbol(unspecified) -> K) .
eq k(apply(fbuiltin(cons), V1, V2) -> K)
 = k((V1, V2) -> makeConsCell -> K) .
eq k((V1,V2) -> makeConsCell -> K) nextLoc(N)
 = k((V1,V2) -> assignToLoc(locs(N, 2))
     -> cell({loc(N) . loc(N + 1)}) -> K) nextLoc(N + 2) .
eq list2Values(cell({L1 . L2}), (Mem [L2,V2]))
 = if (V2 == symbol(nil)) then Mem[L1]
   else ((Mem[L1]), list2Values(V2, (Mem [L2,V2]))) fi .
eq list2Values(symbol(nil), Mem) = nill .
eq list2Names(cell({L1 . L2}),(Mem [L1,symbol(X)][L2,V]))
 = (X, list2Names(V, Mem [L1, symbol(X)][L2, V])) .
eq list2Names(symbol(X), Mem)
 = if (X == nil) then () else (&rest, X) fi .
```

**Fig. 1.** List Operations

cells, excepting the creation of simple constants and variables. For example, in `(define x 4) (display x) x 4`, the `x` and the `4` are not contained in cons cells; they also have no effect on the output (though they are "executed" by K-Scheme).

Figure 1 shows the Maude definitions for the list operations `cons`, `car`, `cdr`, `set-car!`, `set-cdr`, and `cons`. The presence of `apply(mbuiltin(X),V1, V2...)` or `apply(fbuiltin(X),V1 ,V2...)` denotes the application of a built-in syntactic keyword or built-in function to the values `V1`, `V2` ..., respectively.[2] The constructor `cell` accepts a cons cell as an argument and creates a value, i.e., {`L1 . L2`}, is a cons cell, while `cell({L1 . L2})` is a value. `Mem[L]` "returns" the value `L` points to in the store `Mem`. Note that operator `k` wraps the continuation where all computation happens, and that the `K` variable matches the *rest of the continuation*. The constructor `symbol` is to symbols what `cell` is to cons cells (it converts a symbol into a value); the same is true for any other type constructor.

The equations defining the semantics of the two `set` functions place `symbol(unspecified)` on the continuation because this is the return value of the `set` functions. We decided to have a literal unspecified value in places where R$^5$RS declares the result to be unspecified. It is thus possible to have a list

[2] The difference between syntactic keywords and built-in functions is that all of the values passed to a function are pre-evaluated, while those to a syntactic keyword are not. This is necessary for constructs such as `if`.

of unspecified values which, when printed, looks like `(#<unspecified> ...)`. What the `set` equations say, then, is: take the value `V`, assign it to the location in the cons cell, and return the unspecified value as a result to the rest of the computation (the continuation).

When `cons` is applied, we use the `makeConsCell` operator (several other equations in the definition need to create cons cells, so the complexity is factored out). The equations for `makeConsCell` are given in Figure 1 as well. Note that because we actually assign to locations, the `nextLoc` attribute described in Section 3.2 is modified. The operator `assignToLocs` has been defined to allow for the assignment of multiple values at a time.

The last equations shown in Figure 1 are for `list2Values` and `list2Names`. These are for converting between the Scheme style lists, and flat `ValueList`'s and `NameList`'s. `ValueLists` are necessary for passing to procedures (among other things), while `NameList`'s are used for the parameter names for user defined procedures (see Section 3.6). To understand why we need to convert a Scheme list into a `ValueList`, consider the situation we find in a normal application of a procedure in Scheme. The application is simply a list, where the `car` is the procedure, and the `cdr` is the values passed as arguments to the procedure (e.g., `(foo 3 4 5)`). To actually apply the function it is necessary to pull those values out of the Scheme list. This is the job of `list2Values`. The `if_then_else_fi` operator is defined in the Maude prelude (by two trivial equations); `nill` is the identity element for `ValueList`'s.

The most interesting feature of `list2Names` is the way in which improper lists are handled. Before the last name in an improper list, we insert the name `&rest` (inspired by LISP). This signals to the procedure application equations that a variable number of arguments is to be expected (see Section 3.6).

### 3.4 Vectors and Strings

When defining vectors and strings, we again rely on the Scheme storage model. As R$^5$RS mentions "A string... denotes as many locations as there are characters in the string.... A new value may be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before." Logically, strings in Scheme are nothing more than vectors of characters with a special literal syntax. Thus, in K-Scheme, the equations for strings and vectors look very much alike.

Both strings and vectors are defined using what we refer to as the location array, defined as follows:

```
sort LocationArray .
op nill : -> LocationArray .
op [_;_] : Nat Location -> LocationArray .
op __ : LocationArray LocationArray -> LocationArray
        [assoc comm id: nill] .
```

The operator `nill` is the identity operator for location arrays. The structure of each entry in the location array is given by operator `[_;_]`. What this means

is that each entry is a pair of natural number with location, which we use to map natural numbers (vector or string indices) to locations in the store. The last operator specifies the associative and commutative concatenation of entries, so that a single array can be made up of multiple entries. It is commutative because we want to keep the rules for looking up an item in the location array simpler. It is associative because, logically, concatenation of location array cells is irrespective of concatenation order.

Figure 2 shows operations on vectors. Strings, while defined, are omitted from this paper due to their close resemblance to vectors. Both vectors and strings consist of a location array and a natural number denoting the length of the string or vector in question.

```
eq k(apply(fbuiltin(vector-set!),
     vector([N ; L1] LA, I), int(N), V) -> K)
 = k(V -> assignToLoc(L1) -> symbol(unspecified) -> K) .
eq k(apply(fbuiltin(vector-ref),
     vector([N ; L] LA, I), int(N)) -> K) mem(Mem)
 = k(Mem[L] -> K) mem(Mem) .
eq k(apply(fbuiltin(vector-length), vector(LA, I)) -> K)
= k(int(I) -> K) .
eq k(apply(fbuiltin(make-vector), int(I), V) -> K)
 = k(makeVector(V, 0, I) -> K) .
eq k(apply(fbuiltin(make-vector), int(I)) -> K)
 = k(makeVector(symbol(unspecified), 0, I) -> K) .
eq k(makeVector(V, 0, I) -> K) nextLoc(M)
 = k(V -> assignToLoc(loc(M)) -> vector([0 ; loc(M)], I)
     -> makeVector(V, 1, I) -> K) nextLoc(M + 1) .
eq k(vector(LA,I) -> makeVector(V,I,I) -> K) nextLoc(M)
 = k(vector(LA,I) -> K) nextLoc(M) .
eq k(vector(LA,I) -> makeVector(V,N,I)  -> K) nextLoc(M)
 = k(V -> assignToLoc(loc(M))
     -> vector(LA [N ; loc(M)], I)
     -> makeVector(V, N + 1, I) -> K)
   nextLoc(M + 1) .
```

**Fig. 2.** Vector Operations

The equation for `vector-set!` looks much like those for `set-car!` and `set-cdr!` from Figure 1. The difference is that we must find the location mapped, in the location array of the vector, to the index number specified in `vector-set!`. The index is specified as the second parameter to `vector-set!`, matched by `int(N)` in the equation. Likewise, `vector-ref` is highly reminiscent of `car` and `cdr`, again, the only difference being the commutative lookup in the location array. Because we store the length of the vector in the vector, and it is computed when the vector is formed, `vector-length` does nothing more than "returning"

the length value stored in the vector. `make-vector` allows for the creation of vectors of a given length with an optional initial value. R$^5$RS states that if no initial value is specified then the value of each vector element is unspecified. To handle the unspecified case we again use the literal value `symbol(unspecified)`. The first two equations match these two cases: where `make-vector` is called with an initial value, and where it is not. Each of these cases defers the work to the operator `makeVector`. The first parameter to `makeVector` is the initializer value, so for the unspecified case this value will be `symbol(unspecified)`. The next two parameters denote the current index being created and the length of the vector, respectively. The last three equations define the operation of the operator `makeVector`. The first is the start case, when makeVector is first encountered: it creates a new vector value (the `vector` constructor) and maps a location to index 0, simultaneously, it assigns the initializer value to the newly mapped location. The second equation is the termination case: if the index equals the length, the whole vector has been allocated, so we drop the `makeVector` operator and "return" the newly completed vector value. The last equation handles the inductive step.

### 3.5 Quote, Quasi-Quote, Unquote, and Unquote-splicing

In a language with a unified representation of code and data it is important to have some way to distinguish data. In Scheme this is handled via `quote` and its cousin `quasiquote`. In K-Scheme, evaluation of any list is performed by appending the operator `evalk` after that list in the continuation (See Section 3.9). The equation for `quote` is straightforward:

```
eq k(apply(mbuiltin(quote), V) -> K) = k(V -> K)  .
```

As can be seen, application of the syntactic keyword `quote` results in moving its argument onto the top of the continuation. Because it is not succeeded by the operator `evalk` (unless the rest of the continuation matched by the variable `K` deliberately contains `evalk`), the list is not evaluated. R$^5$RS allows for an implementation to use the same memory for all references to a `quote`'d expression, thus we do not need to duplicate the value in this equation.

**Application of Quasiquote, Unquote, and Unquote-splicing** The equations for `quasiquote`, `unquote` and `unquote-splicing` can be seen in Figure 3. Recall that in Scheme `quasiquote` has the shortcut syntax of "`‘`", `unquote` of "`,`", and `unquote-splicing` of "`,@`". The operator qq is placed on the continuation whenever an application of the syntactic keyword `quasiquote` is specified in a program, as can be seen in the first equation in Figure 3. It is also important to bind `unquote` and `unquote-splicing` to their names before evaluation of the `qq` operator, as we do not bind them at initialization time. The reason for this is observed behavior in the interpreters we analyzed: they would call `unquote` and `unquote-splicing` unbound variables if used outside of a `quasiquote` expression. Our whole definition is built around binding built-in functions and

macros to their names. This way, the names can be redefined by users, as allowed in Scheme. The environment is restored after the reduction of `qq` in order to remove the bindings for `unquote` and `unquote-splicing`. The operator `kenv` restores the environment to its argument when it is matched as the top of the continuation.

The presence of `unquote` applied to a value *during evaluation* (the second equation in Figure 3), means that said value must be *evaluated* before being placed into the list (or vector). Thus the value matched by the variable `V` is placed on the continuation followed by `evalk`. Also, we unbind `unquote` and `unquote-splicing` in order to produce the observed behavior of references to unbound variables. What this means is that a program such as `` `(,3) `` will result in `(3)` while `` `(,,3) `` will result in an unbound variable (`unquote`) error. Again we use the `kenv` operator to restore the environment after the evaluation of `V`, in this case rebinding `unquote` and `unquote-splicing`.

`unquote-splicing` is very similar to `unquote`. The difference is that after the `evalk` we place the operator `uqs`. This operator will be matched by later equations in order to know that the value preceding it should be *spliced* into the list (or vector).

```
op qq : Value Nat -> ContinuationItem .
op uqs : -> ContinuationItem .
eq k(apply(mbuiltin(quasiquote), V) -> K) env(Env)
 = k((mbuiltin(unquote), mbuiltin(unquote-splicing))
     -> bindTo(unquote, unquote-splicing)
     -> qq(V, 0) -> kenv(Env) -> K) env(Env) .
eq k(apply(mbuiltin(unquote), V) -> K)
   env(Env [unquote, L1] [unquote-splicing, L2])
 = k(V -> evalk
      -> kenv(Env [unquote, L1] [unquote-splicing, L2])
      -> K) env(Env) .
eq k(apply(mbuiltin(unquote-splicing), V) -> K)
   env(Env [unquote, L1] [unquote-splicing, L2])
  = k(V -> evalk
      -> kenv(Env [unquote, L1] [unquote-splicing, L2])
      -> uqs -> K) env(Env) .
```

**Fig. 3.** Quasiqoute, Unquote, and Unquote-splicing

**Definition of Operator `qq`** The equations in Figure 4 define the operator `qq`. As mentioned in R$^5$RS, "Quasiquote forms may be nested. Substitutions are made only for unquote components appearing at the same nesting level as the outermost back-quote." In order to achieve this, `qq` keeps track of the current nesting depth as a natural number (the Nat in the o `qq` operator declaration).

`unquote` is only evaluated when the nesting level of operator `qq` is 0. The equations for `unquote-splicing` are essentially identical to those for `unquote`, so we do not list them. The equations are broadly separated between application of `qq` to cons cells, and all other values. In the complete K-Scheme, there are also special equations for vectors. The equations for vectors are very similar to the equations for `make-vector` in Section 3.4, the only difference being that the equations essentially copy an already existing vector, and that `qq` is applied to each of the values in the vector being copied. The only particular caveat is that splicing a list into a vector requires inserting each list element into the vector. We have an equation for all non-list and non-vector values, so that the addition of a new value type to the definition will not require a modification to the equations for `qq`.

The first equation in Figure 4 corresponds to a `quasiquote` expression within a `quasiquote` expression (a nested `quasiquote`). Because this occurs while copying a list, rather than *evaluating* a list, we must check to see if the `car` of the list is the symbol `quasiquote`, rather than the equations seen previously where we matched the `apply` operator with a specific built-in function or syntactic keyword. This same strategy of looking at the `car` of the cons cell is used in all of the `qq` equations where the argument to `qq` is a cons cell. symbol `quasiquote` we create a new cons cell consisting of the symbol `quasiquote` and `qq` applied to the `cdr` of the cell (as the `cdr` very well could be an `unquote` form, or another `quasiquote`). The `makePair` operator is a wrapper for `makeConsCell`, which we saw earlier. The difference is it evaluates and collects its arguments before reducing to `makeConsCell`. We also increment the nesting depth, ensuring that only the proper number of `unquote` expressions will be evaluated.

The equations for `qq` applied to a cons cell in which the `car` of the cell is `unquote` are next. If the current nesting depth is 0, the cell is simply evaluated. This will result in eventually matching the equations we saw earlier in Figure 3. If, however, the nesting depth is not 0, the nesting depth is decremented, and we repeat the process of `quasiquote` repeated within a `quasiquote` expression. That is, we make a cons cell of the symbol in question (either `unquote` or `unquote-splicing`) and `qq` applied to the `cdr` of the cons cell.

The last case for cons cells happens when the `car` of the cons cell is none of the three symbols we care about (`unquote`, `unquote-splicing`, `quasiquote`). In this case `qq` is applied to both the `car` and `cdr` or the list, and the resulting values are made into a new cons cell via the `makePair` operator.

Finally, is the base case where the value in question is not a cons cell (or vector). In this case the value is simply copied into the list (or vector), by placing the value, as-is, onto the top of the continuation. This is correct because any application of a function can only occur within a cons cell.

**Unquote-splicing Specifics** Figure 5 shows equations for `unquote-splicing`. In Scheme, `unquote-splicing` works much like `unquote`. It evaluates its argument. However, with `unquote-splicing`, the argument must evaluate to a list. This list is then *spliced* into its enclosing structure. In simple terms, parenthe-

```
eq k(qq(cell({L1 . L2}), N) -> K)
   mem(Mem [L1, symbol(quasiquote)])
 = k(symbol(quasiquote) -> qq(Mem [L2], N + 1) -> makePair
     -> K) mem(Mem [L1, symbol(quasiquote)]) .
eq k(qq(cell({L1 . L2}), 0) -> K)
   mem(Mem [L1, symbol(unquote)])
 = k(cell({L1 . L2}) -> evalk -> K)
   mem(Mem [L1, symbol(unquote)]) .
eq k(qq(cell({L1 . L2}), N) -> K)
   mem(Mem [L1, symbol(unquote)])
 = k(symbol(unquote) -> qq(Mem[L2], N + (-1)) -> makePair
     -> K) mem(Mem [L1,
   symbol(unquote)]) .
eq k(qq(cell({L1 . L2}), N) -> K) mem(Mem)
 = k(qq(Mem[L1], N) -> qq(Mem[L2], N) -> makePair -> K)
   mem(Mem) .
eq k(qq(V,N) -> K) = k(V -> K) .
```

**Fig. 4.** Operator `qq`

ses are removed. For example `‘( 1 2 ,@(list 3 4) 5)` results in the list `(1 2 3 4 5)`. As noted before, a given value has resulted from the evaluation of an `unquote-splicing` if it is succeeded by the operator `uqs`.

```
op uqs : Value -> ContinuationItem .
eq k(cell({L1 . L2}) -> uqs -> qq(V, N) -> K)
 = k(qq(V, N) -> uqs(cell({L1 . L2})) -> K) .
eq k(uqs(V1) -> makePair(V2) -> K)
 = k(append(V1, V2) -> V1 -> K) .
eq k(append(cell({L1 . L2}), V) -> K)
   mem(Mem [L2, symbol(nil)])
 = k(V -> assignToLoc(L2) -> K)
   mem(Mem [L2, symbol(nil)]) .
eq k(append(cell({L1 . L2}), V) -> K)
   mem(Mem [L2, cell(C)])
 = k(append(cell(C), V) -> K) mem(Mem [L2, cell(C)]) .
```

**Fig. 5.** Equations Specific to Unquote-splicing

The first equation in Figure 5 simply reorders the continuation so that evaluation can happen. At this point we have a cons cell at the top of the continuation resulting from the application of `unquote-splicing`. We know that it must be a cons cell, because the result of an `unquote-splicing` form must be a list. The reordering takes the `qq` operator after the `uqs` and places it at the top of the continuation. The cell in front of `uqs` is placed as the argument of a second `uqs`

operator. Both of these operators were defined in Figure 3. The second equation matches the latter `uqs` operator, which takes a value as an argument (and this value must be a cons cell). The presence of `uqs(V1)` followed by `makePair(V2)` (another `makePair` operator defined in our infrastructure), denotes that a cons cell should be made from `uqs(V1)` and `V2`. `uqs` signals that `V2` should be *appended* to `V1`, rather than the normal action of forming a cons cell. In this case, appending means replacing the `symbol(nil)` at the end of `V1` with `V2`. `V1` *must* end with a `symbol(nil)` as the argument to `unquote-splicing` must result in a proper list. The right hand side of the equation, then, says to modify `V1` in place by appending, then "return" `V1` to the continuation. The last two equations simply handle the case of recursively traversing `V1` until `symbol(nil)` is found, and then replacing the symbol by `V2` via assigning to the location in the cons cell that originally pointed to `symbol(nil)`. The first of these equations is the base case (where the `cdr` is `symbol(nil)`), the second is the inductive step (where the `cdr` is anything else).

### 3.6 Lambda

```
op fclosure : NameList ValueList Env -> Value .
eq k(apply(mbuiltin(lambda), V, VL) -> K) mem(Mem)
   env(Env)
 = k(fclosure(list2Names(V, Mem), VL, Env) -> K) mem(Mem)
   env(Env) .
eq k(apply(fclosure(Xl, &rest, X, VB, Env'), VL) -> K)
   env(Env)
 = k(values2List(restN(VL, length(Xl))) -> bindTo X
     -> firstN(VL, length(Xl)) -> bindTo Xl
     -> apply(mbuiltin(begin),VB) -> kenv(Env) -> K)
   env(Env') .
eq k(apply(fclosure(Xl, VB, Env'), VL) -> K)  env(Env)
 = k(VL -> bindTo Xl -> apply(mbuiltin(begin),VB)
     -> kenv(Env) -> K)
   env(Env') .
eq k(apply(mbuiltin(begin), V, VL) -> K)
 = k(V -> continue -> apply(mbuiltin(begin), VL) -> K) .
eq k(V -> continue -> apply(mbuiltin(begin), nill) -> K)
 = k(V -> evalk -> K)  .
eq k(V -> continue -> K)  = k(V -> evalk -> discard -> K) .
```

**Fig. 6.** Lambda and Begin

The heart of the support for the `lambda` syntactic keyword is the `fclosure` value type. The operator declaration for `fclosure` can be seen in Figure 6, as well as the equations for `fclosure` application and creation. The `fclosure` operator

accepts a `NameList`, a `ValueList`, and an `Env` (environment) as arguments. The `Namelist` is a list specifying the name of parameters; recall that we showed how a Scheme-style list is converted to a `NameList` in Section 3.3. The `ValueList` is the body of the procedure. Each `Value` in the `ValueList` is an expression, so it will either be a cons cell or a simple type such as a variable or an integer. We allow a `ValueList` because the bodies of procedures are allowed to consist of multiple expressions that are to be executed in order, much as expressions given to a `begin` syntactic keyword expression.

The first equation in Figure 6 is for the creation of `fclosure Value`'s via the application of the syntactic keyword `lambda`. The equation says when `lambda` is applied to a `Value V` followed by a `ValueList VL`, convert `V` into a `NameList`, put `VL` into the `fclosure` as the body, and store `Env` as the environment of the closure. The reason this works is the syntax of `lambda`. In a program `lambda` always has the form `(lambda (names ...) body)`. Because of this we can be sure that the first `Value` passed to `lambda` must be a Scheme list consisting of the parameter names. Note that we store only the local environment to the closure, as we stated in Section 3.2. For an example of why we do this consider:

```
(define x 3)
(define y 4)
(define f #f)
(let ((y 3))
  (set! f (lambda () (+ x y)))
)
(f) ===> 6
(set! y 5)
(f) ===> 6
(set! x 5)
(f) ===> 8
```

The idea illustrated here is that the change to `x` is visible to the closure, while changes to `y` are invisible, because the `x` in `f` is global, while the `y` is local. When a variable reference is evaluated the local environment is checked first, the global is only checked if there is no reference to the variable in the local environment.

The next two equations show the application of `fclosure`'s. The first thing to notice is that the environment is set to be that of the one stored in the closure, giving us the desired behavior from the above example. In both cases the values passed to the `fclosure` are bound to the names in the `fclosure`'s name list.

In the second equation we see our special symbol `&rest`. Recall that this symbol is inserted by `names2List` when the specified list is improper, because an improper parameter list is Scheme's way of specifying a variable number of parameters to a procedure. When `&rest` appears in the `NameList` we first convert all the values passed after the `&rest` symbol into a Scheme list and then bind it to the name appearing after the symbol `&rest`. The operator `restN` is a simple operation on `ValueList`'s that takes the last `Value`'s of a `ValueList` from a passed number and returns them as a `ValueList`. In this case we pass the length of the `NameList XL`. We then use the `firstN` operator to pull the

first `Value`'s out of the list, and bind these to the beginning of the `NameList`. In the third equation the situation is much less complicated, the passed `Value`'s are simply bound to the `NameList`. In both equations we apply the syntactic keyword `begin` to the `ValueList` that is the body of the `fclosure`. Using this strategy, all sequences of expressions *not* at the top level (i.e., within an expression), are handled by the `begin` operator. Because of this, in K-Scheme, any procedure or keyword which requires sequential evaluation (e.g., `do`, `let`, `let*`, `letrec`, etc) apply the keyword `begin` in their equation. The last requirement is to restore the environment after executing the body; this is, again, handled by `kenv`.

`begin` itself is relatively simple. All it does is execute each expression in order and discard the result, except for the last expression where the result is placed on the continuation in order to "return" it. `begin` is defined by the final three equations shown in Figure 6. The first of them is the termination case wherein the `ValueList` being evaluated is exhausted. The last of them, which will only be applied if the termination case cannot be, converts the operator `continue` into an `evalk` followed by `discard`. Operator `continue` takes no arguments and is just a convenient marker. `discard` simply throws away whatever `Value` precedes it. Recall that `evalk` is an operator telling the definition to evaluate the `Value` before it, rather than Keeping it as is (see Section 3.9).

### 3.7 Call-with-current-continuation

As one might guess, a semantics based on explicit continuations makes a procedure like `call-with-current-continuation` (`call/cc`) fairly straightforward. The equations for `call/cc` can be seen in Figure 7. We defined another `Value` type called `continuation`. It is basically the same as an `fclosure` save without a parameter list. The current (entire) continuation is saved in the `continuation` with the current environment in the first equation. Because only one expression at a time from the top level is in the `k` operator, this works. The following example showcases why only the current expression should be grabbed by `call/cc`:

```
(define k #f)
(+ 4 (call/cc (lambda (c) (set! k c) 4))) ===> 8
(k 3) ===> 7
```

In this example, if the continuation of the whole program is grabbed, rather than just the continuation of `(call/cc (lambda (c) (set! k c)))`, this program will not terminate. The reason for this is that `(k 3)` will be included in the continuation it is calling! This non-termination can be recreated in K-Scheme, or any correct Scheme implementation, by passing the `call/cc` expression and `(k 3)` to `begin`.

Application of a continuation, as seen in the second equation, simply replaces the current continuation (`K2`) with the one contained in the `continuation` operator. The current environment is also replaced by that of the `continuation`. Note that, unlike `fclosure` application, the existing environment is not saved by continuation application. `V`, the value the `continuation` is applied to, is passed to the remainder of the computation.

K-Scheme also contains definitions for `call-with-values` and `dynamic-wind`. While [15] claims that special consideration for `dynamic-wind` must be made, we use the version presented in [6]. Instead of actually modifying the objects created by `call/cc`, this implementation is written completely in Scheme. It does redefine `call/cc`, but we believe, because it can be written with normal `call/cc`, that actually modifying the structure of `continuation` objects is unnecessary. The version presented in [6] uses internal defines, which we do not support. To avoid internal defines, we use the `letrec` equivalency presented in R$^5$RS.

```
eq k(apply(fbuiltin(call/cc), V) -> K) env(Env)
 = k(apply(V, continuation(K, Env)) -> K)  env(Env) .
eq k(apply(continuation(K1, Env1), V) -> K2)  env(Env2)
    = k(V -> K1) env(Env1) .
```

**Fig. 7.** Call-with-current-continuation

### 3.8 Equivalency Predicates

The `eqv?` function in Scheme is a fairly interesting case. According to R$^5$RS "The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 should normally be regarded as the same object." For the most part this is fairly straightforward: two numbers are `eqv?` if they are equal. The same is true for all simple data types. It is more interesting with complex objects. Cons cells, vectors, and strings are equal if they represent the same locations in the store. Thus:

```
(eqv? "foo" "foo") ===> #f
(define x "foo")
(eqv? x x) ===> #t
```

Interestingly enough, because of the way we defined our `Value` types, this comes out to simple equality of terms of sort `Value`. The `==` operator in Maude (and in all other rewrite engines) performs a (recursive) comparison of its two term (normal form) arguments, modulo corresponding attributes such as `assoc.` and `comm.`. If all the sub-objects are equivalent the objects are equivalent. This is similar to `equal?` in Scheme (explained below). The equation for `eqv?` is in Figure 8. All it does is return `symbol(#t)` if the two `Value`'s are equal. This works, because for two cons cells to be equal as Maude terms, they must define the same locations.[3] This holds for strings and vectors as well. It also holds for `fclosures` with some interesting results. R$^5$RS states that two procedures, when compared via `eqv?` must return `#f` if they represent two procedures with

[3] Recall that a cons cell has the form {`L1 . L2`}.

```
eq k(apply(fbuiltin(eqv?), V1, V2) -> K)
 = k(symbol(if V1 == V2 then #t else #f fi) -> K) .
eq k(apply(fbuiltin(eq?), V1, V2) -> K)
 = k(symbol(if V1 == V2 then #t else #f fi) -> K) .
eq k(apply(fbuiltin(equal?), V1, V2) -> K) mem(Mem)
 = k(if equal(V1, V2, Mem) then symbol(#t)
     else symbol(#f) fi -> K) mem(Mem) .
eq equal(cell({L1 . L2}), cell({L3 . L4}), Mem)
 = equal(Mem[L1], Mem[L3], Mem)
   and equal(Mem[L2], Mem[L4], Mem) .
eq equal(vector([ N1 ; L1 ] LA1, N2),
         vector([ N1 ; L2 ] LA2, N2), Mem)
 = equal(Mem[L1], Mem[L2], Mem)
   and equal(vector(LA1, N2), vector(LA2, N2), Mem) .
eq equal(vector(nill, N1), vector(nill, N1), Mem)
 = true .
eq equal(V1, V2, Mem) = V1 == V2 .
```

**Fig. 8.** Equivalency Predicates

different semantics. It says the results of comparing two equivalent procedures is undefined, however. In K-Scheme procedures with different semantics will not be equivalent in terms of Maude equivalency because they will contain different cons cells as the body. There are a few cases where our `eqv?` will return `#t` for procedures. Either they can be the same physical procedure (i.e., two procedures bound to a variable `x` followed by `(eqv? x x)`), or they can be two procedures with the exact same parameters (and parameter names) with the same simple type as the body, e.g.:

```
(eqv? (lambda (x) x) (lambda (x) x)) ===> #t
(eqv? (lambda (x y z) 3) (lambda (x y z)) 3) ===> #t
(eqv? (lambda (x) x) (lambda (y) y)) ===> #f
(eqv? (lambda (x) '(x)) (lambda (x) '(x))) ===> #f
```

The reason the last example returns `#f` is because `'(x)` is not a simple type, and in each `lambda` expression a different list consisting of only the symbol `x` is allocated.

Aside from behavior on empty strings and empty vectors, `eq?` is only allowed to return `#t` when `eqv?` returns `#t`. The idea, as is mentioned in R$^5$RS, is that `eq` can often be more easily implemented (and thus faster) than `eqv?`, such as with a pointer comparison. Our decision was to use the same implementation for `eq?` and `eqv?`, which conforms to the information specification.

`equal?` is a library procedure that actually recursively compares the elements to complex structures, calling `eqv?` on simpler sub-elements (such as numbers). As can be seen in the third equation in Figure 8, when `equal?` is applied to two `Value`'s we pass these `Value`'s to the operator `equal`. If `equal` returns the Maude boolean true, we return the symbol `#t`, if it returns false, we return `#f`.

The `equal` operator recursively compares complex structures. In the case of cons cells, two cons cells are equivalent if the `car`'s and `cdr`'s are equivalent (equation four). Vectors are equivalent if all elements they point to are pairwise equivalent (equation five). It is identical for strings, as strings are little more than specialized vectors. The last equation catches all simple `Value`'s. Two simple `Value`'s are equivalent if they are equal according to Maude. Since this is what we did for `eqv?` this is the same as applying `eqv?` for the simple `Value`'s; exactly what $R^5RS$ states.

### 3.9 Eval

We have explained the role of the `evalk` continuation item operator in previous sections. This is the motor behind the formal definition for the `eval` procedure. The equation for `eval` can be seen in Figure 9 (the first equation), as well as the equations for evaluating the different `Value` types of K-Scheme.

```
eq k(apply(fbuiltin(eval), V, environment(Env1)) -> K)
   env(Env2)
 = k(V -> evalk -> kenv(Env2) ->  K) env(Env1) .
eq k(symbol(X) -> evalk -> K) = k(X -> K) .
eq k(X -> K) mem(Mem) env([X,L] Env)
 = k(Mem[L] -> K) mem(Mem) env([X,L] Env)  .
eq k(X -> K) mem(Mem) globalenv([X,L] Genv)
 = k(Mem[L] -> K) mem(Mem) globalenv([X,L] Genv) .
eq k(cell(C) -> evalk -> K) mem(Mem)
 = k(preApply(list2Values(cell(C),Mem)) -> K) mem(Mem) .
eq k(fclosure(Xl, VL, Env) -> evalk -> K)
 = k(fclosure(Xl, VL, Env) -> K) .
eq k(fbuiltin(X) -> evalk -> K) = k(fbuiltin(X) -> K) .
eq k(mbuiltin(X) -> evalk -> K) = k(mbuiltin(X) -> K) .
eq k(int(I) -> evalk -> K) = k(int(I) -> K) .
```

**Fig. 9.** Eval

We can see in the first equation that the application of the procedure `eval` to a `Value V` places `V` on the top of the continuation followed by `evalk`. `eval` also expects an `environment Value` to be passed. Much like an `fclosure`, the environment is swapped for the passed environment and restored via `kenv` after execution.

The second equation shows the evaluation of symbols. The `symbol` operator is stripped off. Other equations (the third and fourth) lookup symbols in the store if the symbol is on the top of the continuation. Note, as was mentioned earlier, the local environment is checked first; the global is only consulted if the local has no binding for the given variable.

```
eq k(preApply(symbol(X),VL) -> K)
 = k(X -> preApply(VL) -> K) .
eq k(preApply(V,VL) -> K)
 = k(V -> evalk -> preApply(VL) -> K)
eq k(fbuiltin(X) -> preApply(VL) -> K)
 = k(preEval(fbuiltin(X); VL) -> K) .
eq k(fclosure(Xl,VL,Env) -> preApply(VL') -> K)
 = k(preEval(fclosure(Xl,VL,Env); VL') -> K) .
eq k(continuation(K,Env) -> preApply(VL') -> K')
 = k(preEval(continuation(K,Env); VL') -> K') .
eq k(mbuiltin(X) -> preApply(VL) -> K)
 = k(apply(mbuiltin(X), VL) -> K) .
```

**Fig. 10.** Operator `preApply`

The fifth equation handles evaluation of a cons cell. As Scheme requires, the evaluation of a cons cell is always considered to be a procedure or syntactic keyword application (note that user defined macros are expanded before evaluation). The operator `preApply` (Figure 10) decides how to handle the `ValueList` passed to it. If the first `Value` in the `ValueList` is a symbol the symbol is looked up. If it is not a symbol it must be a literal applicable value. Applicable values consist of `fclosure`, `fbuiltin`, `mbuiltin`, `continuation`, and literal `lambda` expressions. The second equation in Figure 10 is this case where the first `Value` in the `preApply` is not a symbol. It must be evaluated in case it is a literal `lambda` expression. The last four equations in Figure 9 show the evaluation of `fclosure`'s, `fbuiltin`'s, `mbuiltin`'s and `int`'s. All simple `Value` types have similar equations, where they are simply placed onto the continuation as-is. Logically this makes sense, as, in Scheme, `'4` is the same thing as `4`. In the latter, `4` is evaluated, but `4` evaluates to `4`! The implication of this is that whether the first `Value` in the `preApply` is a literal `lambda` or an already evaluated `fclosure` we will eventually arrive at the fourth equation in Figure 10. In the case of `preApply` of procedures and continuations, the `preEval` operator is used. All this operator does is evaluate the rest of the arguments. Once all the arguments are evaluated the `apply` operator, seen before, is placed on the continuation, with the given `fbuiltin`, `fclosure`, or `continuation`. Syntactic keywords directly reduce to `apply` *without* evaluating the arguments (no `preEval`) as shown in the last equation in Figure 10.

### 3.10 Macros

K-Scheme supports the use of top-level `define-syntax` to define new macros. This support is under development, so the types of macros that can be defined are still limited: most macros with list-based patterns can be defined, but patterns with improper lists or vectors are still not supported. Macros are also assumed to not define new names using internal `define`s or reference free-names not defined

at top-level. Even with these limitations, K-Scheme can support a number of standard macros, such as those used to define constructs like `or` and `let`. This provides two definitions of these constructs, one via semantic rules and one based on translation into more basic constructs. Macro expansion happens up front, taking a K-Scheme expression with macros and yielding an expression without. This expansion process is orthogonal to the K-Scheme semantics presented so far.

**Processing Macro Definitions** When a macro definition is encountered, K-Scheme processes each provided pattern, transforming it into a form which can more easily be used during matching. These patterns, along with the associated templates, are then stored in a syntax definition map keyed by name. This allows definitions to be quickly found during macro expansion.

```
 op trans : List List NameList -> List .
ceq trans ( [ X I IL ] , [ IL'           ] , NL )
  = trans ( [   I IL ] , [ IL' patVar(X,0) ] , NL )
   if nameIn(X,NL) == false /\ isEllipses(I) == false .
```

The initial pattern is transformed using the `trans` operator, the definition of which is shown above. `trans` takes the original list, a working list (the post-transformation list), and a list of names. The names are the literals defined in `syntax-rules`, and are used to distinguish literals from pattern variables. The sample equation shows a potential match. Here, a name, `X`, is at the head of the list being processed. If it is not in the list of literal names, checked with `nameIn`, and if the following list item is not an ellipses, checked with `isEllipses`, then `X` is a non-repeating pattern variable, and is marked as such in the working list. The item that represents non-repeating pattern variables, `patVar`, includes the name of the variable and a counter, which represents the ellipses "depth" of the variable; this allows us to detect when the ellipses count between the pattern and the template do not match.

**Macro Expansion** To support macro expansion, all expressions processed by K-Scheme are first checked to determine if they make use of any defined macros. If a macro usage is found, the macro is expanded, replacing it with the generated syntax. The expression is then checked again, with this process repeated until no further expansions occur. This model naturally supports both recursive patterns and the use of multiple distinct macros in an expression. The operators that control this process are shown below:

```
op applySyntax : ExpList SynMap -> ExpList .
op applyToExp : Exp SynMap -> Exp .
op applyOneStep : Exp SynMap -> Exp .
```

The first operator, `applySyntax`, is invoked each time a new list of expressions is processed by K-Scheme. It makes use of `applyToExp` to apply the syntax in the syntax map (`SynMap`) to each expression. `applyToExp` applies one step of syntax transformation using `applyOneStep`, repeating this process until the expression no longer changes.

```
ceq applyToExp(E,SM) = applyToExp(E',SM)
   if E' := applyOneStep(E,SM) /\ E =/= E' .
 eq applyToExp(E,SM) = E .
```

The first equation shows the case where the expression does change, meaning that `E` contained a use of a macro that was then expanded in `E'`. In this case, we continue looking for macros to expand in `E'`. The second equation represents where no changes were found (i.e., where the first equation did not apply). In this case, the expression `E`, now fully expanded, is returned.

**Matching and Substitution** Expansion works using a two step process. In the first step, matching, the expander searches for a pattern that matches the supplied syntax. The list of patterns associated with the macro keyword is tried in order. If a match is found, a mapping from pattern variables to expression syntax is returned. Alternatively, match failure causes the next pattern to be tried in turn. The `match` operation, with a sample equation, is shown below:

```
  op match : List List MatchPairs -> MatchPairsXBool .
  eq match( [ E IL ] , [ patVar(X,N) IL' ] , MPs )
   = match( [ IL ], [ IL' ], MPs { patVar(X,N), E } ) .
```

Here, `match` takes two lists. The first contains the current syntax being processed, while the second contains the pattern. The final parameter is a set of pairs, where each pair is a map of pattern variables to the syntax they are matched to. The final result is this set along with a flag indicating whether matching was successful. The equation shows a sample match. The next term in the pattern to match is a pattern variable, `X`; if the next term in the syntax list is an expression, `E`, the match of `X` to `E` is recorded in the set of matches.

The second expansion step is substitution. Substitution uses the mapping found during matching, along with the template associated with the matched pattern, to expand the macro to the proper syntax. Variables in the pattern are replaced with the expression syntax from the mapping, taking proper account of ellipses. The `subst` operation, with a sample equation, is shown below:

```
  op subst : Exp Exp Nat MatchPairs -> Exp .
 ceq subst( [ X I IL ],[ IL' ], M,
           ({ patVar(X,0), I' } MPs))
   = subst( [   I IL ],[ IL' I' ], M,
           ({ patVar(X,0), I' } MPs))
    if isEllipses(I) == false .
```

The `subst` operator takes a template expression, the first argument, and generates the expanded expression, built up in the second argument and eventually returned. The third parameter is a natural number, used to track expansion properly for repeating names and repeating lists. The final parameter is the set of matches developed using the `match` operation. The equation shows an example of substituting the value matched to a pattern variable in the `match` operation for a pattern variable in the template. Here, if name `X` is encountered, and is not followed by ellipses, and if `X` is also the name of a pattern variable matched

to list item `I'`, `X` is removed from the template list and it's substitution, `I'`, is added to the end of the working list. When `subst` has emptied the template list, it is finished, and will return the working list.

**Example: Or** A standard example of `define-syntax` is the definition of `or`:

```
(define-syntax or
   (syntax-rules ()
     ((_) #f)
     ((_ e) e)
     ((_ e1 e2 e3 ...)
      (let ((t e1)) (if t t (or e2 e3 ...))))))
```

This pattern includes multiple cases and the use of recursion (in the last case). Expansion works as expected: `(display (or))` translates to `[display #f]`, while `(display (or (> 1 2) (> 3 4)))` translates to `[display [let [['t ['> 1 2]]] [if 't 't ['> 3 4]]]]` .

**Example: Let** Another standard example is the definition of `let`, given as:

```
(define-syntax let
 (syntax-rules ()
   ((_ ((X E) ...) B ...) ((lambda (X ...) B ...) E ...))))
```

This pattern includes just one non-recursive case, but the use of repeating pattern variables is more complex than in the case of `or`. Again, expansion works as expected: `(let ((a 5) (b 6)) (display (+ a b)) (display (* a b)))` expands to `[[lambda ['a 'b] [display ['+ 'a 'b]] [display ['* 'a 'b]]] 5 6]` .

**Example: Nested Ellipses** A third example illustrates the use of nested ellipses:

```
(define-syntax test
 (syntax-rules ()
   ((_ (X ...) ...) (list X ... ...))))
```

This pattern "strips off" the surrounding list structure, combining all items into a single list. For instance, `(display (test (1 2 3 4) (5 6) (7) (8 9 10)))` expands to `[display [list 1 2 3 4 5 6 7 8 9 10]]`.

## 4 Comparisons and Related Work

The K technique has been used to define several languages previously. Kool [9][10] is an object oriented language designed to show how object oriented language features can be defined in the K framework. A formal definition of Java [7] given in an earlier rewriting logic semantics style from which K descended also exists. There is also a pre-alpha definition of Prolog using K at [26].

Previous attempts at defining Scheme, or portions of Scheme, also exist. As already mentioned [12] gives a partial denotational semantics of Scheme which misses several features (`dynamic-wind`, `eval`, a "top level", etc.), and is not executable.

[5] attempts a rewriting based approached to an operational semantics for Scheme. Our work inherits nothing from this. [5] does not use a list-like internal representation, most operations being performed directly on the program syntax. In order to support `quote` and `eval`, which is mistakenly called `unquote` (referred to as `eval` in the following), `quote` creates a "frozen" expression, which can be later evaluated by `eval`. This is an incorrect approach because it means that only expressions generated by `quote` can be evaluated by `eval`. Our approach is general and supports the evaluation of arbitrary lists, as it should. We also feel that our evaluation of an internal list representation is more in the spirit of the language. Another problem with [5] is that lists themselves are represented as `ValueList`'s rather than cons cells. This would not allow for sharing of `cdr`'s between lists. This works for the subset defined because list modification was not supported (no `set-car!` or `set-cdr!`). Vectors are also mishandled as `ValueList`'s, when they should be lists of locations. `eqv?` could not be handled properly within this framework either. `quasiquote` was also not supported (and adding support for it would be difficult, due to the lack of proper list representation).

[15] provided an operational semantics of $R^5RS$ Scheme. The main contributions of their paper were a greater completeness than the formal definition given in $R^5RS$ (they added `eval`, `quote`, and `dynamic-wind`), modeling multiple return values in a way that is transparent to the rest of the definition, a model of undefined order of evaluation, and that the executability of their definition. We provide a more complete definition of Scheme, offering definitions of `define-syntax`, `quasiquote`. `unquote`, and `unquote-splicing`. Our `eval`, unlike the definition in [15], also supports the environment parameter mentioned in $R^5RS$. We did not feel that a model of undefined order of evaluation was necessary, in light of [15]. Adding undefined order of evaluation to our definition would be fairly straight-forward if desired, however. It could be implemented by creating new continuations for each `Value` for which the order of operation is unspecified, e.g., in the case of function application, where the order of argument evaluation is not specified, a new continuation could be formed for each argument. After evaluation, the `Value`'s could be collected and bound to the parameter names as appropriate. Multiple return values (only appropriate within the context of `call-with-values`) are transparently handled in our definition, `vals` being a particular `Value` type. As mentioned earlier, we do not feel modification of continuations is necessary to support `dynamic-wind`, because an implementation completely written in Scheme exists in [6]. We also feel our definition is more true to the spirit of the semantics of Scheme, wherein code and data have a unified representation, rather than only using a list structure in the presence of `quote`, `cons`, `list`, etc. (as in [15]).

## 5 Future Work and Conclusions

Eventually, we intend to provide complete support for macros, with `let-syntax` and `letrec-syntax`, as well as support for macros involving improper lists and vectors. We also intend to provide full support for the entire Scheme standard library (excepting input, due to the nature of the Maude implementation).

We have presented a nearly complete, formal definition of $R^5RS$ Scheme. The complete source and an online trial of our definition can be found at [16]. Unlike earlier formal, executable definitions, we provide definitions for `quasiquote`, `unquote`, `unquote-splicing`, and `define-syntax` (with portions of its associated pattern language).

## References


1. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
2. P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and C. Ringeissen. An Overview of ELAN. In *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.3). SRI International, January 2007, http://maude.cs.uiuc.edu, 2007.
5. M. d'Amorim and G. Rosu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.
6. R. K. Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2003.
7. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of Computer-aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 501 – 505, 2004.
8. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. 2000.
9. M. Hills and G. Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of the International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, 2007. To appear.
10. M. Hills and G. Roşu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 107–121, 2007.
11. G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, Lecture Notes in Computer Science, pages 22–39, 1987.
12. R. Kelsey, W. Clinger, and J. R. (eds.). Revised$^5$ Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.
13. C. Kirchner, P. Moreau, and A. Reilles. Formal Validation of Pattern Matching Code. In *Proceedings of Principles and Practice of Declarative Programming*, pages 187–197. ACM Press, 2005.

14. N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
15. J. Matthews and R. B. Findler. An Operational Semantics for R5RS Scheme. In *Proceedings of Workshop on Scheme and Functional Programming*, September 2005.
16. P. Meredith, M. Hills, and G. Roşu. K-Scheme website.
17. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
18. J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 1–44, 2004.
19. J. Meseguer and G. Roşu. The Rewriting Logic Semantics Project. In *Proceedings of Structural Operational Semantics*, volume 156 of *Electronic Notes in Theoretical Computer Science*, pages 27–56, 2006.
20. J. Meseguer and G. Roşu. The Rewriting Logic Semantics Project. *Theoretical Computer Science*, 373(3):213–237, 2007.
21. P. D. Mosses. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.
22. G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Department of Computer Science, University of Aarhus. 1981.
23. G. Roşu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation, 2005 and 2006. Version 1: UIUCDCS-R-2005-2672 and Version 2: UIUCDCS-R-2006-2802.
24. T. Şerbănuţă and G. Roşu. Computationally Equivalent Elimination of Conditions - Extended Abstract. In *Proceedings of Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006.
25. T. Şerbănuţă, G. Roşu, and J. Meseguer. A Rewriting Logic Approach to Operational Semantics – Extended Abstract. In *Structural Operational Semantics*, Electronic Notes in Theoretical Computer Science, 2007.
26. T. Şerbănuţă, R. Sasse, M. A. Turki, and G. Roşu. Mprolog website.
27. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: the ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
28. E. Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.
29. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.