A K 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
http://fsl.cs.uiuc.edu

Abstract. This paper presents an executable rewriting logic semantics of R^{5}RS Scheme using the K definitional technique [19]. We refer to this definition as K-Scheme. The presented semantics follows the K language definitional style but is almost entirely equational. It can also be regarded as a denotational specification with an initial model semantics of Scheme. Equational specifications can be *executed* on common rewrite engines, provided that equations are oriented into rewrite rules, typically from left-to-right. The rewriting logic semantics in this paper is the most complete formal definition of Scheme that we are aware of, in the sense that it provides definitions for more Scheme language features than any other similar attempts. The presented executable definition, K-Scheme, can serve as a platform for experimentation with variants and extensions of Scheme, for example concurrency. K-Scheme also serves to show the viability of K as a definitional framework for programming languages.

Key words: Programming language semantics, Rewriting logic semantics, Term 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 [9]) gives a thorough but informal description of the language, as well as a partial denotational semantics. The denotational semantics in [9] 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 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 executability.

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

Recent attempts have been made at giving formal, operational/executable semantics to fragments of Scheme [11, 3]. Unfortunately, the partial definition in [3] does not use a proper representation for vectors and lists, so it cannot be extended to the complete Scheme, and neither [11] nor [3] 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. It should be noted that one motivation of this definition with respect to [11] was to use a completely different definitional style, in addition to defining more. 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 large subset of R^5RS Scheme, called K-Scheme. K-Scheme 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 [19] within rewriting logic [13]. 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. Rewriting logic is a unified logic for concurrency that extends equational logic with transitions; we mostly use the equational fragment of rewriting logic in this paper. Rules are only used to specify non-deterministic features of Scheme (procedure application sub-term evaluation, copying in `quote`, and order of `unquote` expression evaluation in `quasiquote`). One of the driving goals of K-Scheme has been to show the viability of K for defining complex, real world languages, like Scheme. Scheme was chosen particularly for its meta-programming facilities, which provide a strong test for K. We chose to implement this definition directly in Maude [2] using the K-style because, currently, there is no automatic translator from the K notation to Maude, but in this paper we shall show the K notation for the rules we discuss, as they are more compact and easier to understand.

Currently, K-Scheme consists of 802 Maude equations and 7 rules, 192 of them for `define-syntax` macros and 610 for the core of the language (and a few built-in procedures). We define 60 features of Scheme, using 285 auxiliary operators and over 2500 lines of non-comment Maude; 374 lines of code, however, define aspects of the K framework also common to other language definitions, and simple helping operations.

The complete Maude definition of K-Scheme can be found on K-Scheme's webpage at [12], together with a web interface allowing one to "execute" programs directly within K-Scheme's definition, using Maude's capability to execute rewriting logic specifications. The main limitations of K-Scheme at this point are an incomplete standard library 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). Nevertheless, this is the most complete formal definition of R^5RS Scheme of which we are aware. In particular we believe we are first to give for-

mal definitions to the operations of `quasiquote`, `unquote`, `unquote-splicing`, a partial definition of `define-syntax`, as well as a definition of `quote` which represents partial copying.

On Rewriting Logic Semantics and K. This paper is part of the rewriting logic semantics (RLS) project (see [16, 14] 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* [13]. It has been shown in [20] that conventional definitional styles, such as big-step [8] and small-step SOS [18], MSOS [17], reduction semantics with evaluation contexts [22], 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 [19] 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, it can be 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.

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 [13, 10] organizes term rewriting modulo equations as a complete logic and serves as a foundation for programming language semantics [14–16]. 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: $l \to r$. 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, θ, 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. 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 λ calculus where λ terms can be grouped into equivalence classes based on relations such as α and β equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency (non-determinism), 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 K: A Computation-based Rewriting Logic Semantics

K [19] is a rewriting logic semantics framework consisting of a technique and a specialized notation, to define programming languages as rewriting logic theories. By K, we understand the K definitional technique within rewriting logic. Rules in K are denoted by a solid line where rewriting takes place, while equations are denoted by a dotted line. Only the part above the line is replaced, and the special symbol $\cdot$ denotes nothing.

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 *Computation*, 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 $\curvearrowright$. The computation 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 K one wants to schedule for processing/evaluating expression E, all one needs to do is replace the current computation in the state configuration by $E \curvearrowright K$. After E evaluates to *Value* V the computation will be $V \curvearrowright K$.

Lists, used frequently in K, have special matching syntax, due to the frequency of matching against them.

$$k(V_1 \curvearrowright \underset{\cdot}{\underline{V_2}}\rangle \tag{2.3.1}$$

$$k\langle \underset{\cdot}{\underset{\cdot\cdot\cdot}{V}}) \tag{2.3.2}$$

2.3.1 is an example of a *rule* which removes the second item in the *Computation* list. 2.3.2 is an *equation* which removes the last item in a list. In each case the angle bracket denotes that the list continues in the direction of the bracket. Lists are associative data structures, K also frequently makes use of associative *and* commutative data structures, i.e. multisets. Multisets can use the same special list matching syntax, but one must keep in mind that an equation or rule will attempt to match modulo commutativity. The *Computation* k, mentioned above, is an example of a list used in K, where $\curvearrowright$ represents concatenation. *Environments*, which map *Names* to *Locations*, and *Stores*, which map *Locations* to *Values*, are examples of multisets of pairs.

3 Scheme in K

In K-Scheme we attempted to cover the *entirety* of core Scheme as defined, informally, in R^5RS [9]. 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.

3.1 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 [18]. The rewrite rules require this state structure to determine the context of equation application. The major concerns are that all needed 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 current computation (which contains only the current "top level" expression[1] , not the entire program); *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 3.6); *output*, the output of the program modified by calls to `display`; *program*, the stored syntactic representation for the rest of the program not currently in the computation; and *unquotes*, a list of all the *Values* to be `unquote`'d in a given `quasiquote` expression.

3.2 Lists

In K-Scheme, 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 [9]. 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`.[2]

$$\underset{Mem[L_1]}{k(\underbrace{apply(fbuiltin(\texttt{car}), cell(\{L_1.L_2\}))}\rangle mem(Mem)} \tag{3.2.1}$$

$$\underset{Mem[L_2]}{k(\underbrace{apply(fbuiltin(\texttt{cdr}), cell(\{L_1.L_2\}))}\rangle mem(Mem)} \tag{3.2.2}$$

$$\underset{V \curvearrowright assignToLoc(L_1) \curvearrowright symbol(unspecified)}{k(\underbrace{\quad apply(fbuiltin(\texttt{set-car!}), cell(\{L_1.L_2\}, V)) \quad}\rangle mem(Mem)} \tag{3.2.3}$$

Fig. 1. List Operations

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 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 `car`, `cdr`, and `set-car!`, other operations are similar and omitted due to space constraints. The presence of $apply(keyword(X), V_1, V_2...)$ or $apply(fbuiltin(X), V_1, V_2...)$ denotes

[1] We refer to both statements and expressions simply as expressions.
[2] An example of this, "share-cdr", can be seen on [12].

the application of a built-in syntactic keyword or built-in function to the *Values* V_1, V_2 ..., respectively.[3] 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* (*symbol* does the same for *Names*). *Mem[L]* "returns" the *Value L* points to in the store *Mem*.

The equation defining the semantics of the `set-car!` function places *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 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). We consider having a literal unspecified *Value* to be a faithful depiction of R^{5}RS, as K-Scheme alerts the user to a case where Scheme defines the answer to be unspecified. Note that, while K-Scheme has support for vectors and strings, the definitions are removed due to space constraints.

3.3 Lambda and Procedure Application

The keyword `lambda` and procedure application are integral to Scheme.

$$k(\underset{fclosure(list2Names(V,Mem),VL,Env)}{\underbrace{apply(keyword(\texttt{lambda}),V,VL)}}\rangle mem(Mem)env(Env) \tag{3.3.1}$$

Equation 3.3.1 defines application of the keyword `lambda`. When `lambda` is applied to *Value V* and a *Value* list *VL*, *V* represents the parameter list (as a cons cell), while *VL* is the body of the procedure. *list2Names* converts a Scheme list into a list of names. Note that we store the current *Environment* so that it can be restored when the function is applied.

Procedure application is a little more complicated, because R^{5}RS requires that the order of evaluation of the terms in a an application form be non-deterministic.[4]

Equation 3.3.2 defines what happens when we `eval` a list (*evalk* denotes that the *Value* preceeding it need be evaluated). *list2Values* transforms a Scheme list into a K list of *Values*, because it is easier to work with. The evaluation of a list will either be an application of a *continuation*, *function*, or *keyword*. A keyword must be matched early (Equation 3.3.3), as its arguments should not be evaluated. Equation 3.3.4 matches any case where the `cons` of the list is not a keyword. If the `car` is not a keyword we must non-deterministically evaluate the sub terms of the form, i.e. *VL*. The operator *app* denotes that this

[3] The major difference between syntactic keywords and built-in functions in K-Scheme is that all of the *Values* passed to a function are pre-evaluated, while those to a syntactic keyword are not (i.e. lazy evaluation). This is necessary for constructs such as `if`.

[4] The example "nd" on [12] shows this non-determinism in action.

$$k(\underset{preApply(list2Values(cell(C),Mem))}{cell(C) \curvearrowright evalk}\rangle mem(Mem) \tag{3.3.2}$$

$$\underset{apply(keyword(X),VL)}{k(preApply(keyword(X),VL)\rangle} \tag{3.3.3}$$

$$k(\underset{randomEval(unEvalWrap(VL),length(VL)) \curvearrowright app}{preApply(VL)}\rangle \; otherwise \tag{3.3.4}$$

$$k(\underset{V \curvearrowright evalk \curvearrowright randomEval(VL_1,hole,VL_2,N-1)}{randomEval(VL_1,unEval(V),VL_2,N)}\rangle \tag{3.3.5}$$

$$\underset{randomEval(VL_1,V,VL_2,N)}{k(V \curvearrowright randomEval(VL_1,hole,VL_2,N)\rangle} \tag{3.3.6}$$

$$\underset{apply(VL)}{k(randomEval(VL,0) \curvearrowright app\rangle} \tag{3.3.7}$$

$$(\underset{WrongTypeToApply(V)}{apply(V,VL)}\rangle \; if \; \neg isApplicable(V) \tag{3.3.8}$$

$$k(\underset{VL \curvearrowright bindTo(XL) \curvearrowright apply(keyword(\texttt{begin}),VB) \curvearrowright kenv(Env_2)}{apply(fclosure(XL,VB,Env_1),VL)}\rangle env(\underset{Env_1}{Env_2}\rangle \tag{3.3.9}$$

Fig. 2. Function Application

should lead to application, because *randomEval* is also used for evaluating the `unquote` expressions in a given `quasiquote`. The operator *unEvalWrap* simply wraps every *Value* in the *Value* list *VL* with the operator *unEval*. Rule 3.3.5 shows the non-deterministic evaluation of sub terms using *randomEval*. A *Value* which has not been evaluated yet (and is thus wrapped with *unEval*) is chosen and scheduled for evaluation. Its place is kept by the operator *hole* so that it can be placed in the correct place in Equation 3.3.6, which will be matched immediately after *V* is finished evaluating. Because *V* is placed back without the operator *unEval*, it cannot be evaluated again. The natural number *N* is used to know when all the *Values* have been evaluated (Equation 3.3.7). Equation 3.3.8 signals an error if a non-applicable type is the first *Value* in an *apply* operator. Finally, in Equation 3.3.9 we see the final application of an *fclosure*, note that, though we support variable argument number, the equation is not shown. When an *fclosure* is applied, the arguments *VL* are bound to the parameter list *XL*, the *Environment* is set to that of the closure, the body of the lambda is evaluated (`begin` is applied because the bodies of procedures are allowed to consist of a list of expressions), then the *Environment* is restored to the point before the *fclosure* application via the operator *kenv*.

3.4 Eval, Quote, Quasi-Quote, and Unquote

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`. The function `eval`, on the other hand, allows for meta-programming by allowing for the evaluation of lists as programs. Note, that because all code in K-Scheme is represented internally as lists, `eval` is achieved essentially for free. The application of `eval` to any list is treated as an application form, and evaluated accordingly[5].

On the other hand, `quote`, which causes a list to be treated as data, has an interesting under specification in R^{5}RS. R^{5}RS suggests that all `quoted` data be allocated before the program is run, in essence sharing a single `quote`'d list between all uses. It does not, however, *require* this. The alternative would be copying the `quoted` expression for each use. However, these two choices do not encompass the standard, which allows for any combination of copying and sharing.[6]

$$\frac{k(\underdot{apply(keyword(\texttt{quote}), V)}\rangle}{dup(V)} \tag{3.4.1}$$

$$k(\frac{dup(cell(\{L_1.L_2\}))}{dup(Mem[L1]) \curvearrowright dup(Mem[L2]) \curvearrowright makeConsCell}\rangle mem(Mem) \tag{3.4.2}$$

$$k(\frac{dup(cell(\{L_1.L_2\}))}{dup(Mem[L1]) \curvearrowright Mem[L2] \curvearrowright makeConsCell}\rangle mem(Mem) \tag{3.4.3}$$

$$k(\frac{dup(cell(\{L_1.L_2\}))}{Mem[L1] \curvearrowright dup(Mem[L2]) \curvearrowright makeConsCell}\rangle mem(Mem) \tag{3.4.4}$$

$$k(\frac{dup(cell(\{L_1.L_2\}))}{Mem[L1] \curvearrowright Mem[L2] \curvearrowright makeConsCell}\rangle mem(Mem) \tag{3.4.5}$$

Fig. 3. Quote Semantics

Equation 3.4.1 simply says that when we apply `quote` to a *Value*, apply *dup* to that *Value*. We use *dup* because it is needed in a few other places (such as `unquote-splicing`). The operator *dup* non-deterministically decides whether or not to copy *Values*, based on the four rules seen in Figure 3. *dup* is defined for other types as well, but we only show the rules for lists. In each rule, applying *dup* to an element of a cons cell means maybe copying the rest of it, while not applying dup means that the *Value* will *not* be copied. *makeConsCell* simply builds a cons cell with the two proceeding *Values*.

Unfortunately the definition for `quasiquote` is too long to meet the space constraints, we offer instead a summary of the action, and urge the curious to download the complete Maude definition. `quasiquote` of a vector or list iterates through and collects any `unquote` expression at the *proper depth*. The proper

[5] Application of `eval` to any other type is idempotent, except for symbols. `eval`'ing symbols looks up the *Value* bound to the symbol.

[6] An example of any possible combination of copying and sharing can be found on [12] as "quote-nd".

depth is known by keeping a natural number. Any nested `quasiquote` increments the depth, while an `unquote` decrements the depth. If the depth is 0, the `unquote` expression is placed in the attribute *unquotes*, while the operator *hole* (seen before in function application), is placed in the list or vector. The `unquote` expressions are evaluated after the new list or vector is constructed, using the *randomEval* operator seen in 3.3. After they are randomly evaluated they are then spliced back into the list or vector[7].

3.5 Call-with-current-continuation

We felt that the definition of `call-with-current-continuation` (`call/cc`) must be included in this paper primarily because of how easy it is to express in the K style, while it is difficult to impossible to express in other styles (e.g. natural semantics). The equations for `call/cc` can be seen in Figure 4. We defined another *Value* type called *continuation*. It is basically the same as an *fclosure* save without a parameter list.

$$k(\underbrace{apply(fbuiltin(\texttt{call-with-current-continuation}, V) \curvearrowright K}_{apply(V, continuation(K, Env)}) env(Env) \tag{3.5.1}$$

$$k(\underbrace{apply(continuation(K_1, Env_1), V) \curvearrowright K_2}_{V \curvearrowright K_1}) env(\underbrace{Env_2}_{Env_1}) \tag{3.5.2}$$

Fig. 4. Call-with-current-continuation

Equation 3.5.1 says that when `call/cc` is applied to a *Value* V, pass the current continuation, wrapped in the *continuation* constructor, along with the current *Environment* to V (which is assumed to be a function). Equation 3.5.2 defines what happens when a *continuation* object is applied to a *Value* V. That *Value* is simply passed to the remaining computation stored in the *continuation* object, replacing what was the remaining computation. Also the *Environment* in the *continuation* replaces the current *Environment*. Recall from Section 3.1 that k contains only the current "top level" expression. This is to keep `call/cc` from capturing the whole rest of the program, as it should only capture the current "top level" expression.

K-Scheme also contains a definition for `call-with-values` and an *implementation* of `dynamic-wind`. While [11] claims that special consideration for `dynamic-wind` must be made, we use the version presented in [4]. 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

[7] Example "quasiquote-nd" on [12] shows the non-deterministic nature of `quasiquote` in action. Other `quasiquote` examples can be found as well.

continuation objects is unnecessary. Note, however, that this is not a definition of `dynamic-wind`.

3.6 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`. Currently, macros are not hygienic or referentially transparent. Macro expansion happens up front, taking a K-Scheme syntax 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.

$$trans((\underbrace{X\ I}_{I}\rangle, \langle \underbrace{\quad\cdot\quad}_{patVar(X,0)}), XL)\ if\ \neg nameIn(X, XL) \wedge \neg isEllipses(I) \qquad (3.6.1)$$

The initial pattern is transformed using the *trans* operator, the definition of which is shown above. *trans* takes the original list ($kprefixX\ I$), a working list (the post-transformation list, $\langle\cdot\rangle$), and a list of names (XL). 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:

$$applySyntax ::= ExpList\ Synmap \rightarrow ExpList. \tag{3.6.2}$$

$$applyToExp ::= Exp\ Synmap \rightarrow Exp. \tag{3.6.3}$$

$$applyOneStep ::= Exp\ Synmap \rightarrow Exp. \tag{3.6.4}$$

The first operator, *applySyntax*[8], 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.

$$\underset{\dot{E}_2}{applyToExp(E_1, SM)}\ if\ E_2 := applyOneStep(E, SM) \wedge E_1 \neq E_2 \tag{3.6.5}$$

$$\underset{\dot{E}}{applyToExp(E, SM)} \tag{3.6.6}$$

The first equation shows the case where the expression does change, meaning that E_1 contained a use of a macro that was then expanded in E_2. In this case, we continue looking for macros to expand in E_2. The second equation represents where no changes were found (i.e., where the first equation did not apply). In this case, the expression E_1, 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:

$$match(\underset{\cdot}{(E\rangle}, \underset{\cdot}{(patVar(X, N)\rangle}, \underset{\{patVar(X, N), E\}}{(\ \cdot\ \rangle}) \tag{3.6.7}$$

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 (surrounded by curly braces) 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.

[8] $ExpList\ Synmap \rightarrow ExpList$ means that the operator takes an expression list and a syntax mapping, and returns an expression list, the other operators can be read similarly

The second expansion step is substitution (performed by the *subst* operator). 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. A sample equation using the *subst* operator is shown below:

$$subst((\underbrace{X\, I_1}_{I_1}\rangle, \langle\, \underbrace{\cdot}_{I_2}\, \rangle), N, (\{patVar(X,0), I_2\}))\ if\ \neg isEllipses(I) \tag{3.6.8}$$

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 (N), 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_2, X is removed from the template list and its substitution, I_2, 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.[9]

4 Comparisons and Related Work

The K technique has been used to define several languages previously. Kool [6, 7] 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 [5] 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 [21]. Again, one of our main goals of this project has been to show K's definitional viability by defining a language with heavy meta-programming capabilities.

Previous attempts at defining Scheme, or portions of Scheme, also exist. As already mentioned [9] gives a partial denotational semantics of Scheme which misses several features (`dynamic-wind`, `eval`, a "top level", etc.), and is not executable.

[3] attempts a rewriting based approached to an operational semantics for Scheme. Our work inherits nothing from this. [3] 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. Another problem with [3] is that lists themselves are represented as *Value* lists rather than

[9] Examples of macros can be run on [12].

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 *Value* lists, when they should be lists of *Locations*. `eqv?` could not be handled properly within this framework either. `quasiquote` was also not supported.

[11] provided an operational semantics of R^{5}RS Scheme. The main contributions of their paper were a greater completeness than the formal definition given in R^{5}RS (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 the executability of their definition. We provide a definition of Scheme with more features, offering definitions of `define-syntax`, `quasiquote`, `unquote`, and `unquote-splicing`. Further, their definition of *quote* assumes full sharing of *quote*'d values, rather than the non-deterministic sharing allowed by R^{5}RS that we allow in our definition. Our `eval`, unlike the definition in [11], also supports the environment parameter mentioned in R^{5}RS. Multiple return values (only appropriate within the context of `call-with-values`) are transparently handled in our definition, *vals* being a particular `Value` type in K-Scheme, containing multiple *Values*. We also support an undefined order of evaluation for procedure application and `quasiquote`. 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 [4]. We feel the biggest difference between our respective works is the definitional style. As our motivation was to test K as a framework more so than to provide a "better" definition of Scheme, their definition is less relevant to ours than is [3].

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. This will also entail hygiene and referential transparency. Hygiene can be achieved by variable renaming, while referential transparency will require tagging syntax mappings with syntax environments.

We have presented a formal definition of a significant subset of Scheme R^{5}RS Scheme, using the K definitional style within rewriting logic. The complete source and an online trial of our definition can be found at [12]. 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. 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.

3. M. d'Amorim and G. Rosu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.
4. R. K. Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2003.
5. 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.
6. 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.
7. 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.
8. G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, Lecture Notes in Computer Science, pages 22–39, 1987.
9. R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.
10. N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
11. J. Matthews and R. B. Findler. An Operational Semantics for R5RS Scheme. In *Proceedings of Workshop on Scheme and Functional Programming*, September 2005.
12. P. Meredith, M. Hills, and G. Roşu. K-Scheme website, `http://fsl.cs.uiuc.edu/K-Scheme`.
13. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
14. 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.
15. 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.
16. J. Meseguer and G. Roşu. The Rewriting Logic Semantics Project. *Theoretical Computer Science*, 373(3):213–237, 2007.
17. P. D. Mosses. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.
18. G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Department of Computer Science, University of Aarhus. 1981.
19. 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; K was first introduced in the lecture notes of CS322 in 2003.
20. 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.
21. T. Şerbănuţă, R. Sasse, M. A. Turki, and G. Roşu. Mprolog website, `http://fsl.cs.uiuc.edu/index.php/MProlog`.
22. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.