University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2006-2786

A Rewriting-Based Approach to OO Language Prototyping and Design *

Mark Hills Grigore Roşu
University of Illinois Urbana-Champaign
{mhills,grosu}@cs.uiuc.edu

Abstract

This paper introduces a framework for the rapid prototyping of object oriented programming languages. This framework is based on specifying the semantics of a language using term rewriting and a continuation-based representation of control. The notation used, called K, has been developed specifically for programming languages to overcome limitations in more general rewriting notation, and provides for more compact and modular language definitions. The K notation is used to define KOOL, a dynamic object-oriented language with many features found in mainstream object-oriented languages. The ability to rapidly prototype language features is shown both in the definition of KOOL and in the creation of a concurrent extension to the language.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal definitions, design, theory.

Keywords Semantics, rewriting, object-oriented languages.

1. Introduction

Language design is much more an art than a science. Selecting from the wide range of available language features, designing new features, and choosing appropriate syntax are all important tasks which do not have clear ”best” answers. Sometimes even small decisions in any of these areas can drastically impact the usability, or ”feel”, of a language.

Prototyping, then, becomes very important. Prototypes provide the same advantages to language design that they do during program design. By using the language features, instead of just seeing them on paper, the designer can gain confidence that certain features work well, or discover that some features do not. The designer can also find unexpected areas where a design is unclear, or interacts poorly with other parts of the language. Having a method for prototyping languages that does not require the time and effort involved in modifying a compiler or interpreter can thus greatly assist during language design.

There are many methods that have been devised for language prototyping. Some directly use the semantics of the language, while others do not. Some require embeddings into existing languages, with others providing more generic frameworks for language definitions. We use a framework and methodology we refer to as K, which allows for semantics-based language definitions to be created and then, after translation into a related rewriting syntax, executed on general-purpose term rewriting engines. We believe that K provides a powerful, flexible platform for prototyping new languages and language features.

The remainder of the paper is organized as follows. In section 2, we provide a brief introduction to term rewriting, rewriting logic, and K, with additional pointers to more detailed sources of information for those so interested. Section 3 introduces the KOOL language, a dynamic, object-oriented language designed with K. While this definition shows that a fairly complex language can be designed quickly using K, we emphasize this in Section 4 by showing a concurrent extension to KOOL which provides threads and basic mutual exclusion capabilities. Section 5 discusses language case studies that are part of the rewriting logic semantics project, which includes K, while section 6 discusses other related work, focused on other methods which can be used for language design and prototyping. Finally, Section 7 summarizes and presents some avenues for future work.

* Supported by NSF grants CCR-0234524, CCF-0448501, CNS-0509321.

2. K

The K methodology is based around the concept of term rewriting, a simple, generic, yet powerful method of computation. This section provides a brief introduction to term rewriting, rewriting logic, and the K framework. Term rewriting is a standard computational model supported by many systems; rewriting logic [30, 28] organizes term rewriting modulo equations as a complete logic and serves as a foundation for the rewriting logic semantics of programming languages [31, 32, 33]. K [40] is a specialized extension to rewriting logic semantics overcoming some of the limitations of rewriting logic in the context of programming language semantics and allowing for more concise and modular language definitions.

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, which may contain variables, each of the form:

$$l \rightarrow r$$

A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. 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 subterm 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 subterm, rule, and substitution such that $\theta(l)$ matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. It is also possible for the rewriting process to continue forever, a necessity for emulating computation.

There exist a plethora of term rewriting engines, including ASF [46], Elan [1], Maude [8, 9], the OBJ family [16], Stratego [49],

Tom [36, 25], and others. Rewriting is also a fundamental part of existing programming 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 [30, 28] 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 being part of the same equivalence class of terms, a concept similar to that in the λ calculus with equivalence classes based on α and β equivalence of terms. Rewriting logic provides rules in addition to equations, which can be 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. In our usage, both equational and rewriting logic are first-order.

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 the rules, making them $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term built using that definition and "executing" it. Although this can be done in rewrite systems built around rewriting logic, such as Maude, once oriented the rules can be used even in standard term rewriting engines, some having higher brute-force rewriting performance, such as ASF+SDF (with adjustments made to accommodate the feature sets of each – Maude allows commutative and associative operations, while ASF+SDF just allows associative operations, for instance). While staying in rewriting logic and Maude provides many advantages for formal analysis of programs, including model checking and enumeration of possible execution paths, our focus here will be on evaluation, allowing us to stay at the term rewriting level. Therefore, the K definition of KOOL in this paper translates seamlessly into ordinary rewrite systems that can be executed on any of the existing rewrite engines, not only those that support rewriting logic.

2.3 The K Framework

The K framework developed out of our prior work on providing semantics for programming languages using rewriting logic. It provides a domain-specific variant of rewriting logic that we believe is more appropriate for defining programming languages. We believe in particular the K notation improves on rewriting logic (and the related term rewriting) notation in two related key aspects:

1. K is more *concise* – standard rewriting notation shows the left-hand side of a rule and the entire resulting right-hand side. In cases where parts of the left-hand side are just brought in for context (parts are used in the rule, but are not modified), they need to be copied to the right-hand side unchanged. This can make rules much larger and harder to understand, and can also lead to errors in future modifications when the left-hand side is changed but the right-hand side is not. Ideally, one would only need to indicate what is *changing*, which would then make the rules smaller and easier to understand and maintain. K allows this; term rewriting rules equivalent to K rules are often twice as large (or larger) because of the savings we get from the K notational conventions.
2. K is more *modular* – while standard term rewriting notation requires enough context to match an entire subterm, K allows parts of the subterm to be inferred. This allows the unmentioned parts of the subterm to change without requiring the rule to also change. If these parts represent parts of the computational state (memory, concurrency information, etc), this means that this state can change without requiring a change in the rules – only rules that need the changed state must be changed.

In K notation, changes are represented in rules by putting the changes to the term underneath the changed part of the term, with separating lines. For instance, a rule like this:

$$E_1 E_2 E_3 E_4 \rightarrow E_1 V_2 E_3 V_4$$

would be shown in K as:

$$E_1 \underset{V_2}{E_2} E_3 \underset{V_4}{E_4}$$

Again, this allows the introduction of the necessary context without requiring the unchanged part of the context to be duplicated.

In many cases we deal with sets and lists – since we have a first-order representation, the environment, mapping names to locations in memory, is represented as a set of pairs, $Name \times Location$, instead of as a partial function. Input and output can be represented as lists, where an input operation involves removing the first element from an input list and an output operation involves adding the output element to the end of the output list. Since these scenarios are all common in our rules, we have special notation:

- For lists, angle brackets are used to indicate the rest of the list. Thus, an input list of integers where we are interested in retrieving the first integer, N, from the list, would be shown as $\langle N \rangle$, or N and "the rest of the list" (in the direction the "arrow" points), while an output list of integers would be shown as $\langle N)$, or N and everything before it in the list. Lists are assumed to be associative, allowing us to group parts of the list together as needed, but are not commutative since they are ordered. In general, all lists are formed using the comma operation, and all lists have an identity list element "$\cdot$".
- For sets, angle brackets are also used to indicate the rest of the set. A set of $Name, Location$ pairs, with X standing for a name we are interested in looking up and L standing for a location, would be shown as $\langle (X, L) \rangle$. The brackets going either way are intended to represent everything in the set other than what is named – everything "on either side" of the matched item. Sets are assumed to be both associative and commutative, allowing us to rearrange and group parts of the set together as needed. Set formation uses an operation of the form "$\_\_$", two adjacent underscores, meaning the set is formed by just pushing things up against one another. All sets have an identity element "$\cdot$".

Using the K notation, the K definitional technique is based on a first-order representation of *continuations* [38], in our case lists of tasks separated by $\curvearrowright$ (the one list commonly used that is not comma-separated). This representation allows easy access to the current control context, giving us the capability to grab it when needed and potentially save or replace it, a capability which is essential for defining some of the control-intensive features of standard programming languages such as loop break and continue, exceptions, call/cc, and others.

3. KOOL: A Simple Object-Oriented Language

We here define KOOL, a simple, dynamic object-oriented language, loosely inspired by, but not identical to, the Smalltalk language [17, 4]. KOOL has several core features, familiar from other object-oriented languages: all values are objects; all operations are carried out via message sends; message sends use dynamic dispatch; single inheritance is used, with a designated root class named `Object`; methods are all public, while fields are all private outside of the owning object; and scoping is static, yet declaration order for

Program	$P ::=$	$C^*\ E$
Class	$C ::=$	class X is $D^*\ M^*$ end | class X extends X' is $D^*\ M^*$ end
Decl	$D ::=$	var $\{X,\}^+$;
Method	$M ::=$	method X is $D^*\ S$ end | method X ($\{E,\}^+$) is $D^*\ S$ end
Expression	$E ::=$	$X \mid I \mid F \mid B \mid Ch \mid Str \mid (E) \mid$ new $X \mid$ new X ($\{E,\}^+$) | self $\mid E\ X_{op}\ E' \mid E.X(())^? \mid E.X(\{E,\}^+) \mid$ super() | super.$X(())^? \mid$ super.$X(\{E,\}^+) \mid$ super($\{E,\}^+$)
Statement	$S ::=$	E <- E'; | begin $D^*\ S$ end | if E then S else S' fi | if E then S fi try S catch $X\ S$ end | throw E ; | for X <- E to E' do S od | while E do S od | break; | continue; | return; | return E; | $S\ S' \mid E$; | typecase E of Cs^+ (else S)$^?$ end
Case	$Cs ::=$	case X of S

$X \in$ *Name*, $I \in$ *Integer*, $F \in$ *Float*, $B \in$ *Boolean*, $Ch \in$ *Char*, $Str \in$ *String*, $X_{op} \in$ *Operator Names*

Figure 1. KOOL Syntax

classes and methods is unimportant (all methods in a class see all other methods in the same class, for instance, and all classes see all other classes). KOOL is not defined with concurrency features in this section, but is extended to support concurrency in Section 4.

KOOL includes support for standard imperative features, such as assignment, conditionals, and loops with break and continue, as well as features found in many OO languages such as exceptions and run-time type inspection of objects via a typecase construct. Message sends are specified in a Java-like syntax except for methods named after operators, which are always binary and can be used infix (such as `a + b` instead of `a.+(b)`). Because of this, very few operators are predefined.

Sends with no parameters do not require parens except for calls to parent constructors which do not take parameters, which are of the form `super()`. The syntax of KOOL is shown in Figure 1. The lexical definitions of literals are not included in the figure to limit clutter, but are standard (for instance, booleans include both true and false, strings are surrounded with double quotes and characters with single quotes, etc). Single line and block comments are both supported, using the same syntax as JAVA or C++, with the addition that block comments can be nested. Finally, semicolons are used as statement terminators, not separators, and are only needed where the end of a statement may be ambiguous (at the end of an assignment, for instance, or at the end of each statement inside a branch of a conditional, but not at the end of the conditional itself, which has a keyword fi to designate its end).

To get a feel for the language, two sample programs are presented in Figures 2 and 3. In Figure 2, a new class `Factorial` is defined with a method `Fact` that calculates the factorial of the parameter `n`. After the class definition is the main program expression, `console << (new Factorial).Fact(200)`, which creates a new object of class `Factorial`, invokes method `Fact` with the parameter 200, and then writes the output to the predefined `console` object using the output operation, `<<` (borrowed from C++). This operation invokes the `toString` method on any parameters and returns itself as the method result, allowing chaining of output operations (such as `console << "Value = " << 3`).

The second program, in Figure 3, provides a simple example of inheritance and calls to super-methods using a familiar `Point`/`ColorPoint` example. Note that here + is defined in the `String` class as string concatenation.

There is an initial implementation of KOOL available at our website [22], as well as a companion technical report [20] that explains the definition in detail and includes the current code. Programs are parsed using SDF [46] and then executed using Maude [8, 9]. The core of the language is finished (including all semantics discussed here), and we are currently adding additional functionality to the prelude (which includes classes such as `Object`, `Integer`, `String`, and `Console`) as well as using KOOL as a basis for both teaching and research in semantics and OO languages.

```
class Factorial is
  method Fact(n) is
    if n = 0 then return 1;
    else return n * self.Fact(n-1);
    fi
  end
end

console << (new Factorial).Fact(200)
```

Figure 2. Recursive factorial in KOOL

```
class Point is
  var x,y;

  method Point(inx, iny) is
    x <- inx;
    y <- iny;
  end

  method toString is
    return ("x = " + x.toString() + " and y = "
                   + y.toString());
  end
end

class ColorPoint extends Point is
  var c;

  method ColorPoint(inx, iny, inc) is
    super(inx,iny);
    c <- inc;
  end

  method toString is
    return (super.toString() + " and c = " + c.toString());
  end

  method write is
    console << self;
  end
end

(new ColorPoint(20,30,5)).write
```

Figure 3. Inheritance and Built-ins in KOOL

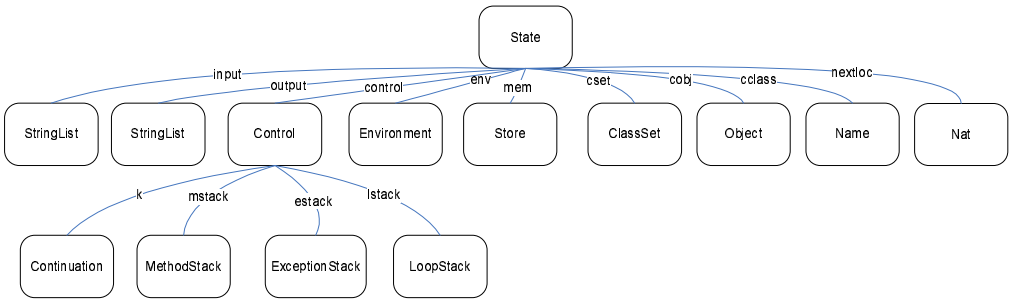


Figure 4. KOOL state infrastructure

3.1 State Infrastructure

One of the key design decisions for a language making use of K is the structure of the state. The K rules make use of this structure to determine the contexts within which the rules are applied, including matching over sets of terms and gathering like elements together into a single subterm that can be manipulated as a whole. It is important then to ensure that all needed information is available and organized into appropriate groups and that the structure is extensible, allowing changes to the semantics that require additions to the infrastructure *without* breaking existing rules in the semantics.

The KOOL state is broken into several distinct pieces, and uses a single explicit layer of nesting to group like components together. A visual depiction of the state is shown in Figure 4.

During program execution, we keep track of names that are in scope and their current memory locations. This is stored in *env*. These memory locations then map to values in *mem*, with the next free memory location in *nextLoc*. We assume garbage collection in KOOL, but do not define it here. Input and output are stored in the *input* and *output* state components, respectively.

Those state components related directly to execution control are stored in *control*. This includes several stacks that are used to quickly recover the program to a state saved at a prior point in time: the method stack (*mstack*), exception stack (*estack*), and loop stack (*lstack*). While not strictly necessary, they save the effort of having to selectively unwind the control context to get back to the proper context for handling a method return or exception catch, for instance. Also included is the current continuation, or *k*, which provides an explicit representation of the current stream of execution and also gives its name to our definitional approach.

Finally, we have several components needed just for the object-oriented features of the language. These include the current object (*obj*) and current class (*class*), which model the object-related portion of the execution context, and the class set (*cset*), which contains information on all classes that have been defined. The class definitions themselves contain information structured as sets, with items representing the class name, parent class name, and sets of methods, among others. Sets are used because of their flexibility; tuples would need to be changed if more information needs to be added to the tuple in an extension to the language, while sets do not, since we can simply match the parts that we want.

3.2 Dynamic Semantics

As with any non-trivial language, there are actually a fair number of K-rules needed to give the semantics of the language. To allow for a fuller explication of the language features in our framework, we have selected several areas that are most illustrative of our technique and are most interesting from an object-oriented perspective. Full details on the language can be found in the companion technical report [20].

The semantics for each area of functionality are separated into individual figures. Of the operators that are used, most are left undefined, since the definition can be derived easily from the context in which the operator is used. For instance, an operator $op(X)$ takes a name as a parameter and, if it is on the continuation, is a continuation item. Thus, it has signature $op : Name \rightarrow ContinuationItem$.

There are two exceptions to this. First, operators are defined for all syntactic constructs in the language in the figure in which they are used. This helps make the leap from the syntax of the language to the semantics. Second, operators are defined if they have attributes, since there would be no other way to know that they have the attributes they have been given. The K attributes in the rules below include *!* and *Memo*.

The first, *!*, specifies that arguments to an operator are automatically put on the continuation for evaluation on top of a new continuation item representing the operator. An example is the `return` E statement, where we want to return the value of the expression E as the result of the message send. Since we need to evaluate E first, we would want to rewrite `return` E to $E \curvearrowright$ `return`, and then actually do the return once E evaluates to a value. This is a common enough scenario that having *!* saves us from having to write a number of rules that all follow this standard form.

The second, *Memo*, just says that we should "Memo-ize" the result of the computation, a feature available in many rewriting systems. This can be seen as an optimization, and does not really impact the semantics. It does, however, allow us to structure the rules in ways that are more natural, without being concerned about having to introduce optimizations into the form of the rules (such as having subterms in the rules that are used to cache results).

When possible, in the rules that follow we make heavy use of matching across contexts. This generally keeps the rules shorter and allows us to focus only on the important elements of the rule and context, without needing to navigate explicitly across intervening parts of the state. Also, rules are over a slightly more abstract version of the syntax, the main differences being that all message sends are transformed into dot notation with explicit (even if empty) parameter lists and terminating semicolons are dropped. All language syntax is presented in a sans serif font, while semantics are presented in *italics*.

3.2.1 Common Operations

Figure 5 shows some K definitions that are used in the definitions of other features, the first two being common to all the programming languages we have defined so far in K. The first contextual rule, Rule (1), defines how the value (V) corresponding to a location (L) is retrieved from memory, when the lookup operation is the next task on the continuation. Note the "$\rangle$" angle bracket to the right of the continuation, saying that the rest of the continuation does not matter here, and the "$\langle$" and "$\rangle$" brackets used to "extract" the pair (L, V) from the store, saying that it does not matter what

$$k(\underset{V}{\underline{lookup(L)}})\ mem\langle(L, V)\rangle \tag{1}$$

$$k(\underset{\cdot}{\underline{V \curvearrowright assign(L)}})\ mem\langle(L, \underset{V}{\underline{\_}})\rangle \tag{2}$$

$$\underset{C'}{\underline{parent(C, \langle cls\langle cname(C)\ pname(C')\rangle\rangle)}} \tag{3}$$

$$\underset{Xl}{\underline{flds(C, \langle cls\langle cname(C)\ flds(Xl)\rangle\rangle)}} \tag{4}$$

$$\underset{CSet'\ \texttt{Object}}{\underline{getInheritsSet(\texttt{Object}, CSet', \_)}} \tag{5}$$

$$getInheritsSet(\underset{parent(C, CSet)\ \ C}{\underline{\quad C \quad}}, \langle\cdot\rangle, CSet) \tag{6}$$

$$\underset{(C, mthd(mname(X)\ MI))}{\underline{getMthd(X, C, \langle cls\langle cname(C)\ \langle mthd(mname(X)\ MI{:}MthdItms)\rangle\rangle\rangle)}} \tag{7}$$

$$getMthd(X, \underset{parent(C, CSet)}{\underline{\quad C \quad}}, CSet) \tag{8}$$

parent : Name × ClassSet → Name [Memo]
flds : Name × ClassSet → NameList [Memo]
getInheritsSet: Name × ClassSet × ClassSet → ContinuationItem [Memo]
getMthd : Name × Name × ClassSet → ContinuationItem [Memo]

Figure 5. K definitions of common operators

other pairs are in the store (a set of such pairs). Once the value is found, the lookup operation is replaced by the expected value, which is then passed to the rest of the continuation.

Rule (2) has a two-hole context, one identifying the value-to-location-assign task on top of the continuation and the other identifying the pair corresponding to the location in the store; once matched, the assign task is eliminated (hence the use of the identity "·") and the current value at that location is replaced by the assigned value. Note the use of an underscore for the current value – similarly to many functional languages, we don't bother giving this value a name since we will not refer to it elsewhere.

The remaining K-rules in Figure 5 define several operators typical in OO programming language definitions, such as ones for locating the parent class, the fields, or a particular method of a class, or the set of names of classes inherited by a class; the syntax of these operators is defined at the bottom of Figure 5.

Rules (3) and (4) are self explanatory; each class's information is "wrapped" with the constructor *cls*, and all classes are kept as a set in the structure referred to with the state attribute *cset*. To get the set of classes inherited by a given class, we can work our way back through parent classes until we reach the Object class, the root of the class hierarchy. In Rule (6), class name *C* is added to the set and *C* is replaced by *C*'s parent. In Rule (5), where the root of the inheritance tree has been reached, `Object` is added to the set and the set replaces *getInheritsSet* on the continuation. Thus, the set is built up in an iterative fashion and then returned. The definition of *getMthd* is straightforward; since KOOL does not allow overloaded method names and uses single inheritance, it is sufficient to check in each class up to the root for a method with the same name, returning the first found. If no matching method is found, an exception (not shown here) is thrown.

Notice that all four operation declarations in Figure 5 are *memoized*, so as discussed above the results will be saved in case they are needed again. While we could perform some optimizations (such as flattening the class definitions to bring in all "reachable" methods), we believe this allows us to leave rules structured in a more intuitive fashion.

$$\underset{\begin{array}{c}control(k(E)\ mstack(\cdot)\ estack(\cdot)\ lstack(\cdot))\\ env(\cdot)\ obj(\cdot)\ class(\cdot)\ mem(\cdot)\ nextLoc(0)\\ cset(process(Classes))\ input(SL)\ output(\cdot)\end{array}}{\underline{eval(Classes\ E, SL)}} \tag{9}$$

Figure 6. Program Evaluation

3.2.2 Program Evaluation

To evaluate a program in KOOL, the program must be inserted into an initial state on which the rewrite process can be started. The state will then proceed through a number of transitions until it reaches a final state (assuming it terminates), which could represent either an error execution, such as one in which an exception is thrown but not caught, causing the program to crash, or a successful execution, yielding some final output and no further execution steps. This is modeled using an *eval* function, shown in Figure 6. Note that the function takes the program and the program input, and then provides default values for all other state components. The semantics will process all class definitions in the program within the *cset* and execute the program expression. Since there are no features yet in the language that can introduce nondeterminism, a given program will always yield the same final state, with the final result in *output*, if it terminates.

3.2.3 Object Creation

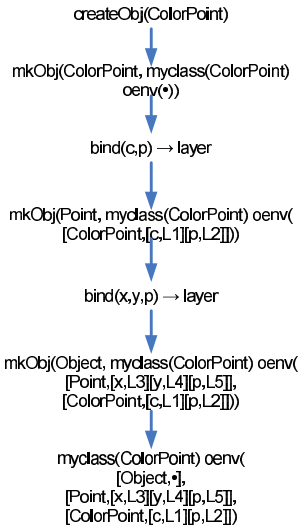
Since all values in KOOL are objects, object creation is one of the core sets of rules in the semantics. At a high level, several distinct steps need to be performed:

- Since each class that makes up the object's type – the current class and all superclasses up to and including `Object` – can contain declarations, and since any of these declarations could be used, depending on the method invoked and the current scope, a "layer" for each class that makes up the object needs to be allocated, containing the layer name and name/location mappings for all instance variables;
- the layers need to be combined into a single object such that lookups occur correctly; specifically, lookups should start at the correct layer, based on the static scoping rules for the language;
- the newly created object, with the various layers and information about its dynamic class, then needs to be returned.

The rules for object creation are shown in Figure 7, along with an example. Rule (10) handles the new expression. new is provided a class name (*C*) and a possibly empty list of arguments (*El*) to be provided to the class constructor. The desired result is that a new object of class *C* will be created and the class constructor for *C*, which must also be named *C*, will be invoked on the newly created object The *createObj* operation indicates that we want to create a new object of class *C*; this is included in a list with the arguments *El* on top of the continuation to make sure these are evaluated as well. The *invokeAndReturnObj* is beneath these waiting for them to yield a list of values; *invokeAndReturnObj* will then cause a method *C* to be invoked on the newly-created object with the values resulting from evaluating *El* passed as the actual parameters. We want to ensure that the object being created is returned at the end of this process; how this is handled can be seen in Rule (11), where *invokeAndReturnObj* is just replaced with an invoke of the same method, a discard to remove the value returned by the method, and finally the target object, effectively replacing the return value of the method with the target object. So, this will take the new object, send it the constructor message with the provided arguments, and return the object, which is what we need. More details about handling message sends are provided in Section 3.2.4.

```
class Point is
  var x,y,p;
  …
end

class  ColorPoint extends Point is
  var c,p;
  …
end
```



$$\frac{\text{new } C(El)}{(createObj(C), El) \curvearrowright invokeAndReturnObj(C)} \quad (10)$$

$$(O, \_) \curvearrowright \frac{invokeAndReturnObj(C)}{invoke(C) \curvearrowright discard \curvearrowright O} \quad (11)$$

$$k(\frac{createObj(C)}{mkObj(C, myclass(C)\ oenv([\texttt{Object}, \cdot])) \curvearrowright Env}) \ env(\underline{Env}) \quad (12)$$

$$\frac{mkObj(\texttt{Object}, O)}{O} \quad (13)$$

$$k(\frac{\cdot}{bind(flds(C, CSet)) \curvearrowright layer} \curvearrowright mkObj(C, \_)\rangle\rangle \ cls(CSet) \quad (14)$$

$$k(\underline{layer} \curvearrowright mkObj(\frac{C}{parent(C, CSet)}, \langle oenv\langle \underline{\cdot} \rangle\rangle\rangle\rangle \ cls(CSet) \ env(\underline{Env}) \quad (15)$$

$$\texttt{new\_(\_)} \ : \ Name \times ExpressionList \rightarrow Expression$$

Figure 7. Object creation

The rules that actually create the object start with Rule (12). As we create each layer, we want to allocate space for any field names which become visible at this layer. By default, this adds the names to the environment. To ensure we don't leak names out, or add names in inadvertently, we first want to save the environment so we can recover it when we are finished and also clear it, so we start with an empty environment and just include field names. This is done by putting the environment *Env* on the continuation (when an environment is encountered at the top of the continuation it is recovered) and setting the *env* state attribute to ·. Also, the *createObj* continuation item is changed to a *mkObj* continuation item, which contains two elements: the current layer that is being built and the object that has been constructed so far. The object, also represented as a set, is initialized with the dynamic class, which matches the class name in the new statement, and a default environment for `Object`, which is empty since `Object` has no fields. We set the current layer being built to the dynamic class of the object, since we need to start with this layer and work up the inheritance tree towards `Object`.

Now, we construct the object in an iterative fashion. Rule (13) shows the base case of the recursive creation, which is when we reach class `Object`. Here, we just take the current object and return this as the result of *mkObj*. Rules (14) and (15) show how the environment layers are configured for classes other than `Object`. In Rule (14), for class *C*, we want to allocate space for all fields in the class and store them in the environment layer assigned to this class in the object being created. To allocate space for the fields, the *bind* continuation item is used. This item is defined to take a list of names, add the names to the environment, and allocate storage for each name. Since there are no values on top of the *bind*, each name will be assigned the initial value nil in the store. *flds* is used to retrieve the fields of class *C*, as defined in class set *CSet*. The *layer* continuation item then says that a new layer should be formed from the resulting environment.

The process of forming this layer is shown in Rule (15), where the current environment is added into the object definition as *[C,Env]*, or the environment layer associated with class *C*. The environment is then cleared out, and the process is continued with the parent class of *C*. Eventually this will reach Rule (13), return the object, call the constructor, and yield a new, initialized object.

In summary, for each layer, we grab back the fields available in the class for that layer. We then allocate space for them, and initialize them to nil. Finally, we save this environment, which just contains the field names and memory locations for this layer, into the object environment (*oenv*), tagging them with this layer's name, then clear out the environment and continue by adding a layer for the parent.

Rules (3) and (4) show the process of getting the parent class and the fields for a given class and class set, respectively.In Rule (3), the class name is used to match against the parent class name in the set representing the class, while in Rule (4) the class name instead matches against the list of names representing the fields of the class.

An example object creation can be seen in Figure 7. The class, `ColorPoint`, contains two fields, `c` and `p`. It extends class `Point`, which contains three fields, `x`,`y`, and `p`. This class extends `Object` by default, which has no fields. As can be seen in the Figure, the continuation item *createObj(ColorPoint)* will lead to the continuation item *mkObj* with the initial class and an initial version of the object. Each step will then either bind fields from the class or add those fields as a new layer in the object environment. Note that there are two copies of field `p`, one at location `L3` and one at location `L5`. The copy chosen will depend on the method being executed – a method from class `Point` will use the copy of `p` at `L5`, while a method from class `ColorPoint` will use the copy of `p` at `L3`. Once the creation reaches `Object`, the new object has been created and is returned. The next step, sending the `ColorPoint` message with the constructor arguments, is not shown.

3.2.4 Message Sends

Message sends are by default dynamic in KOOL. Because of this, lookups for the correct method to invoke should always start with the dynamic class of the object, working back up the inheritance tree towards the `Object` class. There are two exceptions to this rule. First, with `super` calls, the correct instance of the method to call should be found by starting the search in the parent class of the current class in the execution context (in other words, the parent class of the class which contains the currently executing method). Second, with constructor calls, the lookup order is the same, but the method name will change, since constructor method names match

$$\frac{E.X(El)}{(E, El) \curvearrowright invoke(X)} \quad (16)$$

$$\frac{k((myclass(C)\ O), Vl) \curvearrowright invoke(X) \curvearrowright K)\ mstack(\frac{\cdot}{(Env \curvearrowright K, Ctrl, O', C')}\rangle\ Ctrl{:}Control)\ env(\underline{Env})\ obj(\frac{O'}{myclass(C)\ O})\ class(\underline{C'})\ cset(CSet)}{Vl \curvearrowright getMthd(X, C, CSet) \curvearrowright invoke} \quad (17)$$

$$\frac{k(Vl \curvearrowright (C, mthd\langle mparams(Xl)\ mdecls(Xl')\ mbody(K')\rangle) \curvearrowright invoke)\ class(\underline{\_})}{bind(Xl, Xl') \curvearrowright K'} \quad (18)$$

$$\frac{(k(V \curvearrowright \underline{return \curvearrowright \_})\ mstack(\underline{(Ctrl, K, O, C)}\rangle\ \underline{\_ : Control})\ obj(\underline{\_})\ class(\underline{\_})}{K} \quad (19)$$

$\_.\_(\_)$: *Expression* × *Name* × *ExpressionList* → *Expression*
return$_$ *: Expression → Statement [!]*

Figure 8. Message send rules

the class name in which they are defined. The first exception is part of the semantics for `super`, which are separate; the second is part of the core `send` semantics, but is not shown here. The rules for message sends are shown in Figure 8.

The first rule, Rule (16), is used to start processing the message send. The message target, *E*, and the message parameters, *El*, are evaluated, with the name of the message, *X*, saved in the *invoke* continuation item. In Rule (17), given the result of the evaluation of *E* and *El*, the current stream of execution from the continuation (*K*), the control state (*Ctrl*), and the current environment (*Env*), object (*O'*), and class (*C'*) are pushed onto the method stack (with the environment on top of the remaining continuation, so it will be recovered when this continuation is run), ensuring that the current execution context can be quickly restored when the method exits. The continuation is changed to put the value list (*Vl*) that resulted from evaluating the message parameters on top of the *getMthd* continuation item, which is on top of a different *invoke* continuation item that takes no parameters. This indicates that we want to find the method to invoke, based on the method name, class name, and class set, and then invoke it with actual arguments *Vl*. The environment is cleared to ensure names in the current environment aren't introduced into the environment of the executing method, the current object is replaced with the object the message target evaluated to, and the current class is replaced with the dynamic class of the target object (stored in the *myclass* attribute of the object), forcing method lookup to start in the dynamic class.

Rule (18) shows the result of finding the method. A pair of the class name in which the method was found and the method itself are on top of the *invoke* continuation item. This will be replaced with a bind of the method parameters and declarations (*Xl,Xl'*), followed by the method body (*K'*). The values in *Vl* will then be bound to the names in *Xl*, with the declarations *Xl'* bound to *nil*, giving us the proper starting state for executing the method body (by default declarations are assigned a value of `nil` until they are assigned into). The class context is changed to the class, *C*, in which the method was found.

Rule (19) shows the result of reaching the end of a method. All methods are automatically ended with a "return nil;" statement when they are preprocessed, so even method bodies without an explicit return will eventually encounter one. When return is encountered, the *return* continuation item and the rest of the continuation following *return* are discarded, replaced by the continuation on the method stack. The rest of the control state, the current object, and the current class are also reset to the values from the method stack. This will set the execution context back to what it was at the time the message was sent – back to the context of the invoking object. The value on top of the continuation is left untouched, however, since this will be returned as the result of the message send.

3.2.5 Exceptions

KOOL includes a basic exception mechanism similar to that in many other OO languages, such as JAVA or C++. Code can be executed in a `try` block, which has an associated `catch` block. When an exception occurs, control is transferred to the first `catch` block encountered as the execution stack is unwound. The exception, represented in KOOL as an object, is bound to a variable associated with the `catch`, with different classes of exceptions used for different exception conditions (nil reference, message not supported, etc.). Along with system-defined exceptions, custom exception classes can be created, and both can be thrown using a `throw` statement. The semantics for Exceptions can be seen in Figure 9. One important point is that exceptions are not just added by the programmer – they are used in the language semantics as well. For instance, although not shown in the rules for message sends, several possible exceptions can be raised, including an exception generated when a nil variable is used as a message target and an exception thrown when a target object does not support a message (the name and arity must match those in the call). An example where an exception is thrown by the semantics rules can be seen in Figure 12 in Section 4, where an exception is thrown on a lock release when the lock was not already held.

Rule (20) shows the semantics for a try-catch statement. The current control context (*Ctrl*), environment (*Env*), object (*O*), and class (*C*), along with an exception continuation, are all put onto the exception stack. The exception continuation is made up of a binding to the name *X* from the catch clause, the statement *S'* associated with the catch clause, the current environment *Env* (so we recover the current environment and remove the binding of the caught exception to *X*), and the current continuation, *K*. Finally, the try-catch block is replaced with the statement (*S*) from the try clause and the *popEStack* continuation item. So, for a try-catch block, we will execute the statement in the try clause. If this finishes, we will pop the exception stack and continue running. If an exception is thrown, we will instead want to execute the catch clause, binding the exception to the name in the clause, running the body of the catch, and then continuing with the remainder of the computation after the end of the try-catch statement.

The left-hand side of Rule (21) handles the no exceptions case, where the pop marker is found during normal execution. In this case, the top of the exception stack is popped, but no other changes occur. When an exception is thrown, the right-hand side of Rule (21) is used. In this case, the current context information is replaced with the information that was saved on the exception stack, and the exception stack is popped, essentially "unrolling" the execution stack in one shot. The value *V* that represents the exception is left on top, which will cause it to be bound correctly to the catch variable and made available to the catch statement (in Rule (20) the top of the stored continuation was a *bind*, so the value will be

$$(k(\frac{\mathsf{try}\ S\ \mathsf{catch}\ X\ S'\ \mathsf{end}}{S \curvearrowright \mathit{popEStack}} \curvearrowright K)\ \mathit{estack}(\frac{\cdot}{(\mathit{Ctrl}, \mathit{Env}, O, C, \mathit{bind}(X) \curvearrowright S' \curvearrowright \mathit{Env} \curvearrowright K)}\rangle\ \mathit{Ctrl{:}CtrlState})\ \mathit{env}(\mathit{Env})\ \mathit{obj}(O)\ \mathit{class}(C) \quad (20)$$

$$k(\frac{\mathit{popEStack}}{\cdot}\rangle\ \mathit{estack}(\frac{\_}{\cdot}\rangle \quad \Big| \quad (k(V \curvearrowright \frac{\mathit{throw} \curvearrowright \_}{K})\ \mathit{estack}(\frac{(\mathit{Ctrl}, \mathit{Env}, O, C, K)}{\cdot}\rangle\ \frac{\_ : \mathit{CtrlState}}{\mathit{Ctrl}})\ \mathit{env}(\frac{\_}{\mathit{Env}})\ \mathit{obj}(\frac{\_}{O})\ \mathit{class}(\frac{\_}{C}) \quad (21)$$

op try_catch_ _end : *Statement* × *Name* × *Statement* → *Statement*
op throw_ : *Expression* → *Statement*[!]

Figure 9. Exception handling rules

$$\frac{\mathsf{typecase}\ E\ \mathsf{of}\ \mathit{Cases}\ \mathsf{end}}{E \curvearrowright \mathit{getInheritsSet} \curvearrowright \mathit{Cases}} \quad (22)$$

$$\frac{o\langle \mathit{myclass}(C)\rangle \curvearrowright \mathit{getInheritsSet}}{\mathit{getInheritsSet}(C, C, \mathit{CSet})}\ \mathit{cset}(\mathit{CSet}) \quad (23)$$

$$\frac{\langle C\rangle \curvearrowright (\mathsf{case}\ C\ \mathsf{of}\ S\rangle}{S} \ (24) \qquad \langle\_\rangle \curvearrowright (\frac{\mathit{Case}}{\cdot}\rangle \ (25) \qquad \frac{\langle\_\rangle \curvearrowright (\cdot : \mathit{Cases})}{\cdot} \ (26)$$

typecase _ of _ end : *Expression* × *Cases* → *Statement*
case _ of _ : *Name* × *Statement* → *Case*

Figure 10. Typecase rules

bound to the name from the catch clause). Since the rest of the computation after the end of the try-catch statement was saved as part of the exception continuation, the computation will continue correctly after the end of the exception handler.

3.2.6 Runtime Type Inspection

KOOL allows the dynamic type of an expression to be checked at runtime using a `typecase` construct. This construct contains a sequence of cases, each with a class name and a statement. If the class name in the case matches either the dynamic class type of the expression or a superclass of the dynamic class type, the statement is executed. Cases are evaluated from top to bottom, with an optional `else` case that always matches. The rules for runtime type inspection are shown in Figure 10.

Since the parsing step can convert the `else` case to a case matching `Object`, we assume in the semantics that there is no longer a designated `else` case. When a typecase is encountered, Rule (22) shows that this is replaced with an evaluation of the expression *E*, on top of the *getInheritsSet* continuation item, followed by the *Cases* that will be checked. When the expression *E* is evaluated to an object value, Rule (23) shows the start of building the set of class names that will be used in the check against the cases. The *getInheritsSet* continuation item is changed to another item with the same name but three parameters, a class name, a set of class names and a set of classes, with the first two parameters set to the dynamic class of the expression result, *C*. The inherits set is built according to the rules in Figure 5.

With the set of classes for the expression calculated, the remaining three rules, Rules (24), (25) and (26), apply sequentially to process the cases. In the first, a matching case is found, so the class name set ($\langle C\rangle$) and the remainder of the cases list are both discarded, replaced by the statement *S* from the matched case. In the second, the case does not match, but there are cases left in the list, so the current case is removed, allowing the next to be tried. In the third, there is no match, and there are no cases left in the list, so both the cases list and the class name set are discarded, allowing control to fall through to whatever was after the case statement. This provides for the intended semantics – the statement of the first matching case (if any) will execute, then control will pick up with the next statement after the end of the typecase.

3.2.7 Primitives

Since all operations are modeled as message sends, there isn't a native way in the language to, for instance, add two numbers, or output a string. Yet, at some point, `5 + 3` actually has to yield `8`. This is done using primitives, a concept similar to that used in Smalltalk. Each class which is used to represent a primitive value, such as `Integer`, contains a field that stores the primitive value. This field can be accessed by the primitive operations to either take out the existing primitive value or put a new one in. For instance, for `5 + 3`, primitive operations would take out the value `5` and the value `3`, add them using the system version of integer addition, create a new `Integer` object, and put the primitive value `8` into the new object's primitive value field. All "system" operations, including input and output, are handled using primitives, providing the programmer with an object-level view of the primitive operations.

4. Extending KOOL with Concurrency

The dynamic semantics from Section 3.2 does not support any concurrent operations – as defined, KOOL is a sequential language, with a single thread of execution. In this section we illustrate the process of extending a language defined with K by adding concurrency support to KOOL. Although there are many features we could add, concurrency seems to be an especially useful feature to prototype. Not only is there increasing interest in adding concurrency as a native (not library supported) feature of languages, but it is also a non-trivial feature with many design options. In fact, the first prototype of concurrency in KOOL was quite different from what is shown below.

To support concurrency, a new statement, spawn, will be added to create new threads; threads will be able to acquire and release locks on specific objects (similarly to the Java language) using acquire and release statements; and accesses to shared memory locations should *compete* – if two threads both assign a value to a shared variable, the resulting value should be nondeterministic, based on the actual execution order of the threads.

With multiple threads, and thus multiple concurrent streams of execution, some of the state components will need to be duplicated. This includes any components which provide context to the current thread of execution: the current object, the current class, the entire control, and the environment. This allows each thread to have enough local information to execute without interfering with the execution of other threads. For instance, threads should not share the current class, since a message send in one thread would potentially interfere with a message send in the other if they did. However, some information, such as the set of classes and the store, will be global to all threads. The grayed sections of Figure 11 represent the changes in the state from Figure 4 to enable concurrency.

The additional syntax and new rules for the dynamic semantics for concurrency in KOOL are shown in Figure 12. It is important to note that, even though we are adding a significant new feature and making significant changes to the state infrastructure, most of the rules are new – very few existing rules need to be changed. Two rules, Rule 27 and Rule 28, are concurrent versions of Rule 1 and 2, respectively, and are changed by simply boxing them, which in

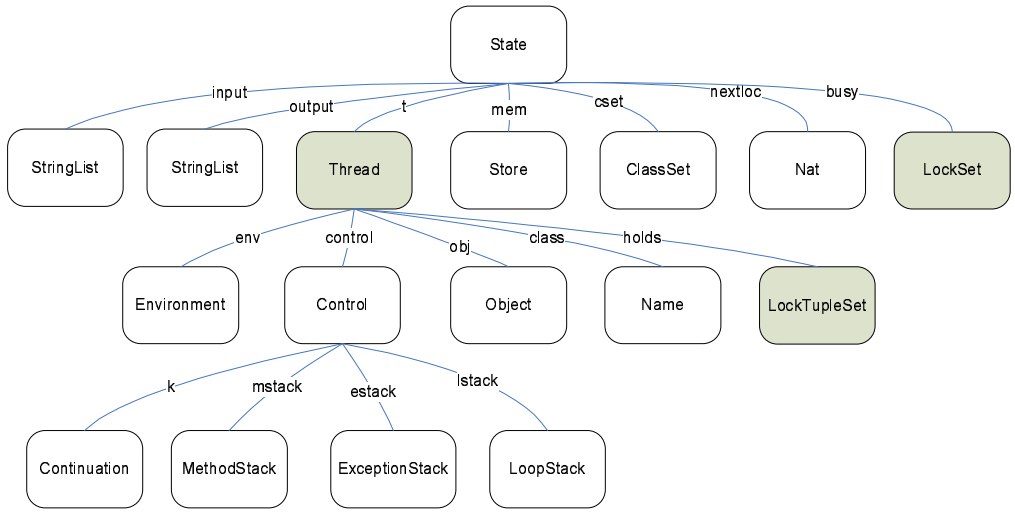


Figure 11. KOOL state infrastructure, with concurrency

$$\boxed{k(\underline{lookup(L)})\ mem\langle(L,V)\rangle \atop V} \tag{27}$$

$$\boxed{k(\underline{V \curvearrowright assign(L)})\ mem\langle(L,\underline{\ \ })\rangle \atop \cdot \qquad\qquad V} \tag{28}$$

$$\frac{eval(Classes\ E, SL)}{newThrd(E,\cdot,\cdot,\cdot)\ mem(\cdot)\ nextLoc(0)\ cset(process(Classes))\ input(SL)\ output(\cdot)\ busy(\cdot)} \tag{29}$$

$$t\langle k(\underline{\mathsf{spawn}\ E})\ env(Env)\ obj(O)\ class(C)\rangle \atop \cdot \qquad\qquad \cdot \qquad\qquad newThrd(E,Env,O,C) \tag{30}$$

$$\frac{newThrd(E,Env,O,C)}{t(control(k(E)\ mstack(\cdot)\ estack(\cdot)\ lstack(\cdot))\ env(Env)\ obj(O)\ class(C)\ holds(\cdot))} \tag{31}$$

$$\underline{t\langle k(\cdot)\ holds(LTS)\rangle} \atop \cdot\ busy(\underline{\quad LS \quad} \atop LS - LTS) \tag{32}$$

$$k(\underline{V \curvearrowright acquire}) \atop \cdot\ holds\langle(V,\underline{\ N\ })\rangle \atop s(N) \tag{33}$$

$$\boxed{k(\underline{V \curvearrowright acquire}) \atop \cdot\ holds\langle\underline{\ \cdot\ } \atop (V,1)\rangle\ busy\langle\underline{\ LS\ } \atop LS\ V\rangle} \Leftarrow V \notin LS \tag{34}$$

$$k(\underline{V \curvearrowright release}) \atop \cdot\ holds\langle\underline{(V,1)} \atop \cdot\rangle\ busy\langle\underline{V} \atop \cdot\rangle \tag{35}$$

$$k(\underline{V \curvearrowright release}) \atop \cdot\ holds\langle(V,\underline{s(N)})\rangle \atop N \tag{36}$$

$$k(\underline{\qquad V \curvearrowright release \qquad} \atop \mathsf{throw\ new\ LockNotHeldEx})\ holds(LTS) \Leftarrow V \notin LTS \tag{37}$$

$\mathsf{spawn\_}$: *Expression* $\rightarrow$ *Statement*
$\mathsf{acquire\_}$: *Expression* $\rightarrow$ *Statement [!]*
$\mathsf{release\_}$: *Expression* $\rightarrow$ *Statement [!]*

Statement S ::= *all prior statements* | $\mathsf{spawn}\ E\ ;$ | $\mathsf{acquire}\ E\ ;$ | $\mathsf{release}\ E\ ;$

Figure 12. Concurrent KOOL rules and added syntax

K indicates that the rule can compete with other boxed rules. One rule, Rule (29), actually does change to take account of the new state infrastructure. This is the concurrent version of Rule 9. Rule 29 makes use of the *newThrd* continuation item to create a new execution thread and set up the starting state appropriately.

The spawn statement creates a new thread based on a provided expression. The expression is evaluated in the new thread, meaning any exceptions thrown by the expression when it is evaluated will be handled in the new, not the spawn'ing, thread. This is a design decision, and has been made to simplify the semantics; the original rule assumed that only method calls could be spawned, and evaluated all method arguments in the current thread, providing different exception behavior. Rule (30) shows the semantics of spawn. Here, the expression E in the spawn statement is given to the *newThrd* item, along with the current environment(*Env*), the current object (*O*), and the current class(*C*). spawn returns no value, so it is just removed from the continuation. Rule (31) shows how the new thread is actually created. The passed values for expression, environment, object, and class are plugged into the proper state components nested within the new thread. This will start the new thread for expression E running in the proper environment. When the thread finishes, it needs to be removed, with any locks it holds being removed from the global busy lock set. This is illustrated in Rule (32).

Along with the ability to create new threads, we also need to be able to acquire and release locks. This is done using the acquire and release statements. The semantics for acquire is shown in Rules (33) and (34). In Rule (33), a lock is acquired on an object V that the current thread already holds a lock on. This just increments the lock count on this object from N to the successor of N. In Rule (34), a lock is acquired on a value V that no thread, including the current thread, has a lock on. This adds the value V and a lock count of 1 to the thread's *holds* set, while also adding V to the current global lock set *LS*. This rule is boxed since multiple lock attempts in situations where no thread already holds a lock on a given object can compete. Also, a lock count is necessary to ensure that lock `acquires` and `releases` are balanced – a thread can acquire a lock multiple times, with a recursive method call for instance, and we need to ensure that a lock is not inadvertently released too soon.

The semantics for release is shown in Rules (35), (36), and (37). In rule (35), a lock on value V with lock count 1 is released. This removes the lock from both the local *holds* set and the global *busy*

```
class ThreadGame is
  var x;

  method ThreadGame is
    x <- 1;
  end

  method Add is
    while true do x <- x + x; od
  end

  method Run is
    spawn(self.Add); spawn(self.Add);
  end
end
(new ThreadGame).Run
```

Figure 13. The Thread Game in KOOL

set. Rule (36) shows what happens when a lock on value *V* with lock count greater than 1 is released – here, the count simply goes from $s(N)$ (the successor of N) to N. Finally, if there is an attempt to release a lock that the thread does not hold, an exception should be thrown. This is shown in Rule (37), where an attempt to release a lock on *V* not held by the thread results in a `LockNotHeldEx` exception being thrown.

A sample concurrent program, the thread game, is shown in Figure 13. In this program a new class, `ThreadGame`, is defined. The constructor for this class sets field `x` to the value `1`. The `Add` method then includes an infinite loop that, during each execution of the loop body, issues a single statement, adding `x` to `x` and assigning the result back to `x`.

If this program were not concurrent, this would just double the value of `x` each time through the loop. However, the `Run` method spawns two threads, each of which will execute the `Add` method. Because there is no synchronization used to prevent data races, the two threads can easily interfere with one another. In fact, it has been proved that the variable `x` can take the value of any natural number greater than 0 [35].

5. Other Language Case Studies

The KOOL language discussed in Sections 3 and 4 illustrates a particular language specification, which is just one example within a much broader language specification methodology, based on a first-order representation of continuations. A key point worth making is that this methodology *scales up* quite well to real languages with complex features, both in terms of still allowing very readable and understandable specifications, and also in being capable of providing high performance interpreters and competitive program analysis tools.

For example, Java 1.4 (see also [7] for a complete formal semantics) and the JVM have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [12, 11]. In fact, the semantics of large fragments of conventional languages are routinely developed by UIUC graduate students taking programming language design and semantics classes, as course or short research projects, including, besides Java and the JVM, languages like (alphabetically), Haskell, Lisp, LLVM, Pict, Python, Ruby, Scheme, and Smalltalk [39].

Typically, one needs two equations or rewrite rules to define the semantics of each language construct: one to *divide* the evaluation task into evaluation subtasks, and the other to *conquer* the task by combining the values produced by the evaluations of the subtasks into a resulting value for the original task. However, there are language constructs that can be translated into other, more general constructs with just one equation (e.g., for loops into while loops), but also language constructs that need many equations or rules. For example, the creation of a new object in Java needs more than 10 equations. Each equation defines a meaningful and different case to analyze, which cannot be avoided or collapsed as a special case of a more general case neither technically nor conceptually; this is due to the inherent complexity of object creation in the presence of inner classes.

A semantics of a Caml-like language with threads was discussed in detail in [31], and a modular rewriting logic semantics of a subset of CML has been given by Chalub and Braga in [6]. Following a continuation-based semantics similar to the one in this paper, D'Amorim and Roşu have given a definition of the Scheme language in [10]. Other language case studies, all specified in Maude, include BC [3], CCS [48, 3], CIAO [44], Creol [23], ELOTOS [47], MSR [5, 42], PLAN [43, 44], the ABEL hardware description language [24], SILF [19], FUN [40], SIMPLE [32], and the π-calculus [45]. Some of these rewrite logic language definitions do not obey the continuation-based style advocated in this paper. That is because those languages lack complex control statements, such as exceptions, or break/continue of loops, or abrupt return from functions, or halt, or call/cc, which the continuation-based style can handle naturally. Nevertheless, those languages can also be given a continuation-based semantics.

6. Other Related Work

There is much related work on defining programming languages in various computational logical frameworks and inside other languages, allowing the defined language to be used for formal analysis, evaluation, and prototyping. We cannot mention all these here, but we refer the interested reader to the rewriting logic semantics project [32, 33, 31] and to the K report [40] for a comprehensive discussion, focused on computational logical frameworks, of the various techniques, comparisons, their advantages and limitations. We here only list a few of them which are, in our view, closer in purpose to our approach.

On the foundational, computational logical side, reduction-based semantics such as SOS [37] and context reduction with evaluation contexts [13] (sometimes called reduction semantics) appear to be quite related to rewriting logic semantics, including K. This is especially true of context reduction. There are, however, several crucial differences. In addition to the complete model-theoretic semantics that endows rewriting logic, the main operational distinction between rewriting logic semantics and reduction semantics is that, in reduction semantics, the applications of reductions are *context-sensitive*, and are sequential in nature (leading naturally to an interleaving semantics for concurrent operations). In rewriting logic semantics they can, and are encouraged to be, applied in an *unrestricted* fashion and *in parallel*, wherever the rewrite rules match, leading to a "true concurrency" semantics for concurrent operations.

Leaving aside the distinctions between the underlying semantics models and the models of concurrency, one could say that, in principle, rewriting logic is a special case of reduction semantics, namely one where contexts are ignored when reduction rules are applied. From this perspective, the complexity and diversity of languages that have been defined so far using rewriting logic [32, 33, 31, 21] can be regarded as empirical evidence that contextual restrictions may be avoided in practice when defining and designing programming languages. The key in achieving this is to apply the reductions not on the original program, but on a transformed, computationally equivalent structure, which is shown in our rules as the continuation wrapped by $k(\cdots)$. Indeed, a structure of the form $k(E \curvearrowright K)$ appearing during the execution of a program can be thought of as $C_K[E]$, where C_K is the "evalua-

tion context" corresponding to K. The advantage of "flattening" and "maintaining" contexts as first order continuation structures in K is that contextual matching is never needed; standard matching and term rewriting alone become sufficient. This opens the door to using standard term rewriting engines, which can achieve remarkable speeds on ordinary machines today – on the order of millions and tens of millions of rewrite steps per second. This is comparable to the speed of conventional programming languages, sometimes very favorably comparable[40]. Other existing tools developed for rewriting logic theories, such as model checkers and path exploration tools, many of which are highly efficient, also become available, providing methods to analyze programs without the need to write tools from scratch.

Turning to more of a focus on language design and prototyping, among the approaches based on term rewriting and related techniques, the first extensive study on defining a programming language equationally, with an initial algebra semantics, was performed in the context of formal reasoning about imperative programs[15]. OBJ [16] was used to execute the language specifications via term rewriting. Interesting work in not only defining languages via term rewriting but also in compiling the definitions has been investigated under the ASF+SDF project [46]. Stratego [49] is a program transformation framework also based on term rewriting. Besides the development and use of the K framework, what distinguishes our work is precisely the use of a first-order representation of continuations and of associative/commutative (AC) matching. Our goals are also somewhat different; ASF+SDF and Stratego have focused more on program transformation and compilation, while OBJ was used to define much simpler languages. We are instead focusing on executability, prototyping and formal analysis of languages with potentially complex features.

Our approach is also similar to that taken by the SECD machine [26], especially in our explicit representation of control and our use of stacks (which are equivalent to SECD dumps). A fundamental difference between the two is that all rules are applied at the top in SECD machines, while rules in K are not similarly restricted, giving us the opportunity to apply rules (for garbage collection, for instance) wherever they match, including in the middle of a continuation. Also, we tend to model more complex languages with more involved configurations (see Figure 11), and our configurations can be dynamic, with new state components added and removed based on operations like thread creation and termination. This leads to natural extensions for concurrency. There are some minor differences as well, including the fact that we keep values directly on the continuation instead of on a separate stack.

There is some similarity between our approach and monads [27, 34]. The monad approach gains modularity at the denotational level by using monad transformers to lift program constructs from one level of specification to a richer one. Haskell was used in an interesting example of building a language interpreter in this monadic style [18]. In our case, modularity is achieved by the use of AC matching and context transformers based on the structure of the state, which allow selecting from the state "soup" only those attributes of interest. The complete enumeration of the state attributes is done only once, when defining the *eval* command.

Other examples define languages inside other, functional or logic languages. One example of this is PLT Redex [29], a Scheme-based tool for defining languages using context reduction. Other examples include definitions of language semantics in Prolog [41], and definitions in functional languages like Scheme [14], where language-level features (Scheme functions and lists, for instance) are used directly to model the language semantics. The Centaur project [2] was very ambitious, allowing definitions in a Prolog-like language and generating language tools and processors. We believe that our independence from a specific language or language implementation is a benefit, allowing us to target rewriting platforms based on feature set and tool support. We also believe this makes the semantics somewhat cleaner, in that we aren't defining a language in terms of another, potentially complex language. However, this comes at the price of not having access to the built-in capabilities of a language such as Scheme, ML, or Prolog.

7. Conclusion and Future Work

In this paper we showed how the K rewrite logic framework and methodology can be used for rapid prototyping, design and experimentation with object-oriented programming languages by defining KOOL, an experimental, sequential object-oriented language, and by then extending KOOL with concurrency. Although this paper has focused specifically on those characteristics that make this technique suited for language prototyping and design, we believe that in general the K-based approach discussed in this paper gives a good balance among often opposite factors such as: mathematical rigor (it is denotational and its initial model semantics is open to inductive reasoning), executability (it is operational by term rewriting), formal analysis, ease of understanding and teaching, tool support, and scalability.

We intend to implement a parser for K and a translator into rewriting logic in the near future. However, as explained in [40], this task is much harder than it may seem and involves researching several important and interesting problems, such as: *sort inference*, because, for elegance and especially for modularity reasons, we'd like to avoid declaring variables whose sorts can be inferred from contexts - this is a non-trivial problem in the context of subsorting and overloading operation names; *tuple operation inference*, because, for the same reasons, we'd like to avoid declaring operations needed only for tupling, such as those placing information in stacks. Also, once a parser is implemented, the next step is to mechanize the compilation technique outlined in [19].

Along with these utilities, we also plan to provide animation support for executions in the semantics, allowing users to view which rules are used in what orders during program executions. We would also like to provide an enhanced environment, potentially based on Eclipse, for working with language definitions. These would be especially useful during the language design process and in classroom settings.

References

[1] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. *ENTCS*, 15, 1998.

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SDE 3*, pages 14–24. ACM Press, 1988.

[3] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*. Elsevier, 2004.

[4] Byte. Issue on Smalltalk. *Byte Magazine*, 6(8), August 1981.

[5] I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*. Elsevier, 2004.

[6] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004.

[7] F. Chen and G. Roşu. Rewriting Logic Semantics of Java 1.4. `http://fsl.cs.uiuc.edu/java-semantics`.

[8] 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.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*,

volume 2706, pages 76–87. Springer LNCS, 2003.

[10] M. d'Amorim and G. Roşu. An Equational Specification for the Scheme Language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005.

[11] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.

[12] A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. In *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 132–147, 2004.

[13] M. Felleisen and R. Hieb. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.

[14] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.

[15] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.

[16] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.

[17] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[18] J. Guy L. Steele. Building interpreters by composing monads. In *Proceedings of POPL'94*, pages 472–492. ACM Press, 1994.

[19] M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *Proceedings of WRLA'06*, ENTCS. Elsevier, 2006. To appear.

[20] M. Hills and G. Roşu. KOOL: A K-based Object-Oriented Language. Technical Report UIUCDCS-R-2006-2779, Computer Science Department, University of Illinois at Urbana-Champaign, 2006.

[21] M. Hills and G. Rosu. FSL Semantics Research Homepage. http://fsl.cs.uiuc.edu/semantics.

[22] M. Hills and G. Rosu. KOOL Language Homepage. http://fsl.cs.uiuc.edu/KOOL.

[23] E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*. Elsevier, 2004.

[24] M. Katelman and J. Meseguer. A rewriting semantics for abel with applications to hardware/software co-design and analysis. In *Proceedings of WRLA'06*, ENTCS. Elsevier, 2006. To appear.

[25] C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In *Proceedings of PPDP'05*, pages 187–197. ACM Press, 2005.

[26] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[27] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of POPL'95*, pages 333–343. ACM Press, 1995.

[28] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[29] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In V. van Oostrom, editor, *Proceedings of RTA'04*, volume 3091 of *LNCS*, pages 301–311. Springer, 2004.

[30] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[31] J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, pages 1–44. Springer LNAI 3097, 2004.

[32] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2006.

[33] J. Meseguer and G. Roşu. The rewriting logic semantics project. In *Proceedings of SOS'05*, volume 156 of *ENTCS*, pages 27–56. Elsevier, 2006.

[34] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[35] J. S. Moore. http://www.cs.utexas.edu/users/moore/publications/thread-game.html.

[36] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of CC'03*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.

[37] G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Dept. of Computer Science, University of Aarhus, 1981.

[38] J. C. Reynolds. The Discoveries of Continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

[39] G. Roşu. Programming language design and semantics classes. Department of Computer Science, University of Illinois at Urbana-Champaign, `http://fsl.cs.uiuc.edu/index.php/Grigore_Rosu#Classes`.

[40] G. Roşu. K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2005-2672, Computer Science Department, University of Illinois at Urbana-Champaign, 2005.

[41] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*. Addison-Wesley, 1995.

[42] M.-O. Stehr, I. Cervesato, and S. Reich. An execution environment for the MSR cryptoprotocol specification language. `http://formal.cs.uiuc.edu/stehr/msr.html`.

[43] M.-O. Stehr and C. Talcott. PLAN in Maude: Specifying an active network programming language. In *Proceedings of WRLA'02*, volume 117 of *ENTCS*. Elsevier, 2002.

[44] M.-O. Stehr and C. L. Talcott. Practical techniques for language design and prototyping. In J. L. Fiadeiro, U. Montanari, and M. Wirsing, editors, *Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing. February 20 – 25, 2005. Schloss Dagstuhl, Wadern, Germany*, 2005.

[45] P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous Pi-calculus semantics and may testing in Maude 2.0. In *Proceedings of WRLA'02*, volume 117 of *ENTCS*. Elsevier, 2002.

[46] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.

[47] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[48] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *Proceedings of WRLA'02*, volume 117 of *ENTCS*. Elsevier, 2002.

[49] E. Visser. Program Transf. with Stratego/XT: Rules, Strategies, Tools, and Systems. In *Domain-Specific Program Generation*, pages 216–238, 2003.