University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2006-2779

KOOL: A K-based Object-Oriented Language *

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

Abstract

This paper documents KOOL, a dynamic, object-oriented language designed using the K framework. The KOOL language includes many features available in mainstream object-oriented languages, including such features as runtime type inspection and exceptions. Since the language is currently changing, this should be seen as a snapshot of the language at this point in time.

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

Keywords Semantics, rewriting, object-oriented languages.

1. Introduction

The K framework is a language-agnostic framework for defining languages using a rewriting-logic approach. Our language, KOOL, is a dynamic, object-oriented language designed using K. At this point it is a research prototype, containing features available in other languages, and is relatively complete. Our goal, however, is to use the language, and the flexibility we gain using the K framework, to explore new language features.

This 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. Section 4 introduces a concurrent extension to KOOL which provides threads and basic mutual exclusion capabilities. Finally, Section 5 summarizes and presents some avenues for future work.

Note that this should be seen as a work in progress. The KOOL language is still changing, and we plan to add additional information to this document both to cover new features of the language and to provide more details on existing language features.

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 [11, 10] organizes term rewriting modulo equations as a complete logic and serves as a foundation for the rewriting logic semantics of programming languages [12, 13, 14]. K [18] 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.

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

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 [19], Elan [1], Maude [3, 4], the OBJ family [5], Stratego [20], Tom [16, 9], 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 [11, 10] 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:

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

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* [17], 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 [6, 2]. 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 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

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	$E ::=$	$X \mid I \mid F \mid B \mid Ch \mid Str \mid (E)$ | new X | new X ($\{E,\}^+$) | self | $E\ X_{op}\ E'$ | $E.X((\,))^?$ | $E.X(\{E,\}^+$) | super() | super.$X((\,))^?$ | super.$X(\{E,\}^+$) | 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; | $S\ S'$ | 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

```
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

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 [8]. Programs are parsed using SDF [19] and then executed using Maude [3, 4]. 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 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

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

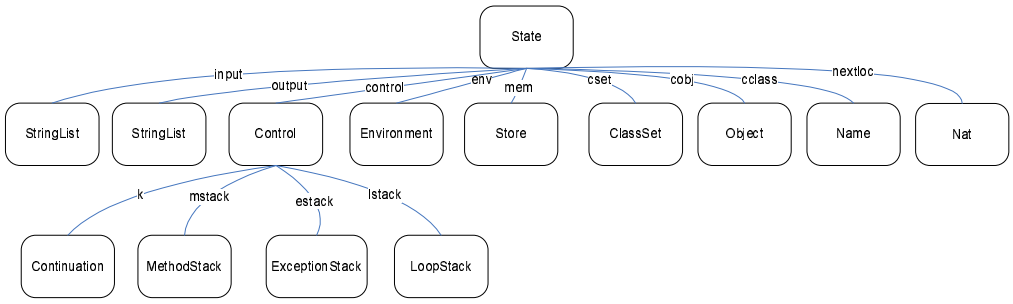


Figure 4. KOOL state infrastructure

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. Documenting these rules in a way that makes them useful (with context, not just a long list) is an ongoing process. Because of this, we have selected a number of areas that are interesting from an object-oriented perspective and that are more complex, making explanations most useful. The full definition of the language, in Maude and (for parsing) SDF, is included in the appendix.

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*.

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

$$k(\underline{V \curvearrowright assign(L)}) \; mem\langle (L, \_)\rangle \quad (2)$$
$$\cdot \qquad\qquad \overline{V}$$

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

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

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

$$getInheritsSet(\frac{C}{parent(C, CSet)}, \langle \cdot \rangle, CSet) \quad (6)$$
$$\overline{C}$$

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

$$\frac{getMthd(X, Object, CSet)}{\texttt{throw new MethodNotFoundException()}} \quad (8)$$

$$getMthd(X, \frac{C}{parent(C, CSet)}, CSet) \quad (9)$$

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

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 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.

$$\frac{\mathit{eval}(\mathit{Classes}\ E, SL)}{\begin{array}{c}\mathit{control}(k(E)\ \mathit{mstack}(\cdot)\ \mathit{estack}(\cdot)\ \mathit{lstack}(\cdot))\\ \mathit{env}(\cdot)\ \mathit{obj}(\cdot)\ \mathit{class}(\cdot)\ \mathit{mem}(\cdot)\ \mathit{nextLoc}(0)\\ \mathit{cset}(\mathit{process}(\mathit{Classes}))\ \mathit{input}(SL)\ \mathit{output}(\cdot)\end{array}} \quad (10)$$

Figure 6. Program Evaluation

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.

Rules (7), (9), and (8) all deal with finding a method that is being search for in the class hierarchy. In Rule (7), the proper method is found in class *C*, and is then placed on the continuation in place of the *getMthd* continuation item along with the class in which it was found. Rule (9) deals with the case where the method has not been found yet; in this case, we want to search for the method in the parent class, so we replace the class we are looking for the method in, *C*, with its parent. If no method of the requested name is found – we reach Object and still have not found it – Rule (8) specifies that we will throw an exception.

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.

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.

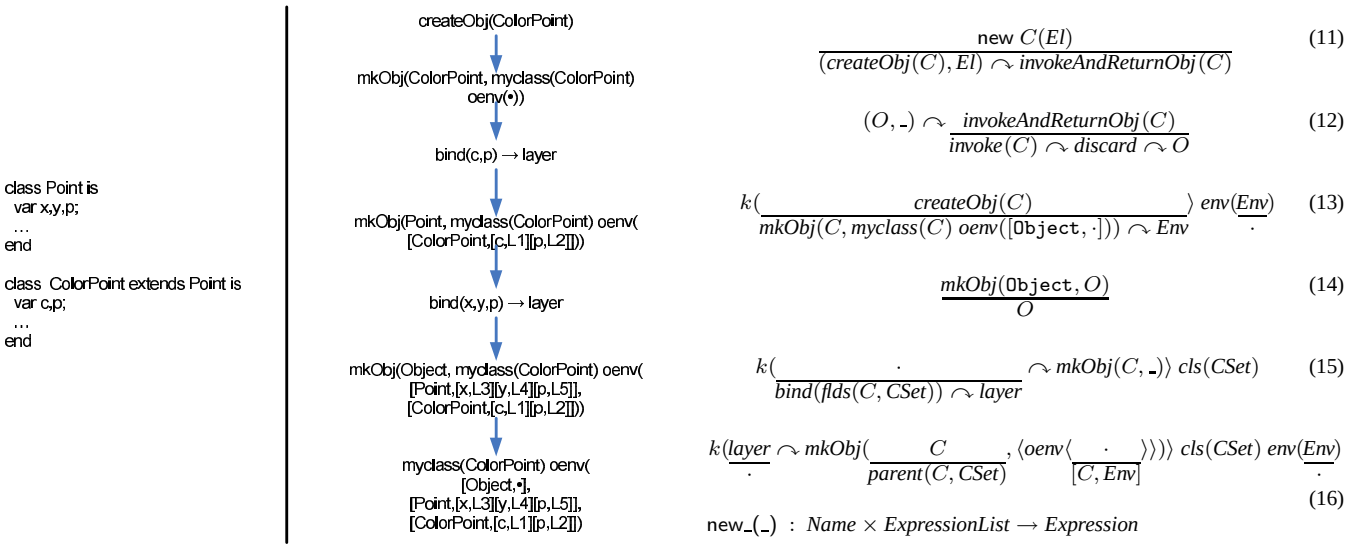
3.2.3 Object Creation

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 (11) 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 (12), 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.

The rules that actually create the object start with Rule (13). 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`.



$$\frac{\texttt{new}\ C(El)}{(createObj(C), El) \curvearrowright invokeAndReturnObj(C)} \quad (11)$$

$$(O, \_) \curvearrowright \frac{invokeAndReturnObj(C)}{invoke(C) \curvearrowright discard \curvearrowright O} \quad (12)$$

$$k(\frac{createObj(C)}{mkObj(C, myclass(C)\ oenv([\texttt{Object}, \cdot])) \curvearrowright Env}\rangle\ env(\underline{Env}) \quad (13)$$

$$\frac{mkObj(\texttt{Object}, O)}{O} \quad (14)$$

$$k(\frac{\cdot}{bind(flds(C, CSet)) \curvearrowright layer} \curvearrowright mkObj(C, \_)\rangle\ cls(CSet) \quad (15)$$

$$k(\underline{layer} \curvearrowright mkObj(\frac{C}{parent(C, CSet)}, \langle oenv\langle \underline{\cdot} \rangle\rangle\rangle\ cls(CSet)\ env(\underline{Env}) \quad (16)$$

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

Figure 7. Object creation

Now, we construct the object in an iterative fashion. Rule (14) 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 (15) and (16) show how the environment layers are configured for classes other than `Object`. In Rule (15), 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 (16), 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 (14), 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 `L2`. 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 the class name in which they are defined. The rules for message sends are shown in Figure 8.

The first rule, Rule (17), 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. Rules for `super` sends are similar, with Rule (18) handling standard `super` calls and Rule (19) handling `super` calls for constructors. Note that, since these are calls on the current object, we don't need to evaluate the target as we do in Rule (17).

In Rule (20), 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 pa-

$$\frac{E.X(El)}{(E, El) \curvearrowright invoke(X)} \tag{17}$$

$$\frac{\mathsf{super}.X(El)}{(El) \curvearrowright superinvoke(X)} \tag{18}$$

$$\frac{\mathsf{super}(El)}{(El) \curvearrowright superinvokeCons} \tag{19}$$

$$k(\frac{(myclass(C)\ O), Vl) \curvearrowright invoke(X) \curvearrowright K)}{Vl \curvearrowright getMthd(X, C, CSet) \curvearrowright invoke}\ mstack(\frac{\cdot}{(Env \curvearrowright K, Ctrl, O', C')}\rangle\ Ctrl{:}Control)\ env(\underline{Env})\ obj(\frac{O'}{myclass(C)\ O})\ class(\underline{C'})\ cset(CSet) \tag{20}$$

$$k(Vl \curvearrowright \frac{superinvoke(X) \curvearrowright K}{getMthd(X, parent(C, CSet), CSet) \curvearrowright invoke})\ mstack(\frac{\cdot}{(Env \curvearrowright K, Ctrl, O, C)}\rangle\ Ctrl{:}Control)\ env(\underline{Env})\ obj(O)\ class(C)\ cset(CSet) \tag{21}$$

$$k(Vl \curvearrowright \frac{superinvokeCons \curvearrowright K}{getMthd(parent(C, CSet), parent(C, CSet), CSet) \curvearrowright invoke})\ mstack(\frac{\cdot}{(Env \curvearrowright K, Ctrl, O, C)}\rangle\ Ctrl{:}Control)\ env(\underline{Env})\ obj(O)\ class(C)\ cset(CSet) \tag{22}$$

$$k(\frac{(nil, Vl) \curvearrowright invoke(X)}{\mathsf{throw\ new\ NilPointerException()}}\rangle \tag{23}$$

$$k(Vl \curvearrowright \frac{(C, mthd\langle mparams(Xl)\ mdecls(Xl')\ mbody(K')\rangle) \curvearrowright invoke)}{bind(Xl, Xl') \curvearrowright K'}\ class(\underline{\_})\ \Leftarrow len(Vl) = len(Xl) \tag{24}$$

$$k(\frac{Vl \curvearrowright (C, mthd\langle mparams(Xl)\ mdecls(Xl')\ mbody(K')\rangle) \curvearrowright invoke\rangle}{\mathsf{throw\ new\ InvalidSignatureException()}}\ \Leftarrow len(Vl) \neq len(Xl) \tag{25}$$

$$(k(V \curvearrowright \frac{return \curvearrowright \_}{K})\ mstack(\frac{(Ctrl, K, O, C)\rangle}{\cdot}\ \underline{\_ : Control})\ obj(\underline{\_})\ class(\underline{\_}) \tag{26}$$

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

Figure 8. Message send rules

rameters. 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.

Again, the rules for super sends, Rule (21) and Rule (22), are similar. Rule (21) is the same as Rule (20) except that we don't change the current object, since it is the message target, and we start the search for the method in the parent class, which is indicated by changing *C* to $parent(C)$ in *getMthd*. We also don't bother changing the class context, since this will be changed if the method is found (and, if it is not found, an exception will be triggered anyway). We did change this in Rule (20), even though it is changed later when the method is found, to keep the state consistent – if we did not, the current object could be of a class completely unrelated to the current class. Rule (22) is the same as Rule (21) except we not only change the second part of *getMthd* to the parent class name, but the first as well, since constructors have names matching their containing class.

Rule (24) 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 (26) 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.

Exceptional cases are handled in Rules (23), (25), and (8). The latter was discussed in §3.2.1, while the other two are new. Rule (23) handles the case where a message target is nil, in which case we raise a NilPointerException. Rule (25) handles the case where we find a method with the name we are looking for, but with the wrong signature (here, the wrong number of arguments). When this happens, we raise an InvalidSignatureException. More information on exceptions can be found in §3.2.5.

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.

$$(k(\underset{S \curvearrowright popEStack}{\underline{\mathsf{try}\ S\ \mathsf{catch}\ X\ S'\ \mathsf{end} \curvearrowright K}})\ estack(\underset{(Ctrl, Env, O, C, bind(X) \curvearrowright S' \curvearrowright Env \curvearrowright K)}{\underline{\cdot}})\ Ctrl{:}CtrlState)\ env(Env)\ obj(O)\ class(C) \tag{27}$$

$$k(\underset{\cdot}{\underline{popEStack}} \curvearrowright \_)\ estack(\underset{\cdot}{\underline{\cdot}} \curvearrowright \_) \quad | \quad (k(V \curvearrowright \underset{K}{\underline{throw \curvearrowright \_}})\ estack((\underset{\cdot}{\underline{Ctrl, Env, O, C, K}}) \curvearrowright \_ : CtrlState)\ \underset{Ctrl}{\underline{\_}}\ env(\underset{Env}{\underline{\_}})\ obj(\underset{O}{\underline{\_}})\ class(\underset{C}{\underline{\_}}) \tag{28}$$

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

Figure 9. Exception handling rules

One important point is that exceptions are not just added by the programmer – they are used in the language semantics as well. For instance, in the rules for message sends in Figure 8, exceptions can be raised when a nil variable is used as a message target or when the number of formal and actual arguments to a method does not match. Another 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 (27) 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 (28) 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 (28) 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 (27) the top of the stored continuation was a *bind*, so the value will be 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 (29) shows that this is replaced with an evaluation of the expression *E*, on top of the *getInheritsSet* continuation item, followed

$$\frac{\mathsf{typecase}\ E\ \mathsf{of}\ Cases\ \mathsf{end}}{E \curvearrowright getInheritsSet \curvearrowright Cases} \tag{29}$$

$$\frac{o\langle myclass(C)\rangle \curvearrowright getInheritsSet\ \ cset(CSet)}{getInheritsSet(C, C, CSet)} \tag{30}$$

$$\frac{\langle C\rangle \curvearrowright (\mathsf{case}\ C\ \mathsf{of}\ S\rangle}{S} \quad (31) \qquad \frac{\langle \_\rangle \curvearrowright (\underset{\cdot}{\underline{Case}}\rangle}{\cdot} \quad (32) \qquad \frac{\langle \_\rangle \curvearrowright (\cdot : Cases)}{\cdot} \quad (33)$$

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

Figure 10. Typecase rules

by the *Cases* that will be checked. When the expression *E* is evaluated to an object value, Rule (30) 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 (31), (32) and (33), 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.

The way that primitives are handled in KOOL is currently changing. In the past, all primitive operations were coded in terms of continuations, requiring them to be coded in the underlying rewrite engine. This also meant the class definitions for classes

$$\boxed{k(\underset{V}{\underline{lookup(L)}})\ mem\langle(L,V)\rangle} \tag{34}$$

$$\boxed{k(\underset{\cdot}{\underline{V \curvearrowright assign(L)}})\ mem\langle(L,\underset{V}{\underline{\_}})\rangle} \tag{35}$$

$$\frac{eval(Classes\ E, SL)}{\begin{array}{c}newThrd(E,\cdot,\cdot,\cdot)\ mem(\cdot)\ nextLoc(0)\\ cset(process(Classes))\ input(SL)\ output(\cdot)\ busy(\cdot)\end{array}} \tag{36}$$

$$t\langle k(\underset{\cdot}{\underline{\mathsf{spawn}\ E}})\ env(Env)\ obj(O)\ class(C)\rangle \underset{\cdot}{\underline{\qquad}} \frac{}{newThrd(E,Env,O,C)} \tag{37}$$

$$\frac{newThrd(E,Env,O,C)}{\begin{array}{c}t(control(k(E)\ mstack(\cdot)\ estack(\cdot)\ lstack(\cdot))\\ env(Env)\ obj(O)\ class(C)\ holds(\cdot))\end{array}} \tag{38}$$

$$\underset{\cdot}{\underline{t\langle k(\cdot)\ holds(LTS)\rangle}}\ busy(\underset{LS-LTS}{\underline{\quad LS\quad}}) \tag{39}$$

$$k(\underset{\cdot}{\underline{V \curvearrowright acquire}})\ holds\langle(V,\underset{s(N)}{\underline{\quad N\quad}})\rangle \tag{40}$$

$$\boxed{k(\underset{\cdot}{\underline{V \curvearrowright acquire}})\ holds\langle\underset{(V,1)}{\underline{\quad\cdot\quad}}\rangle\ busy\langle\underset{LS\ V}{\underline{\ LS\ }}\rangle \Leftarrow V \notin LS} \tag{41}$$

$$k(\underset{\cdot}{\underline{V \curvearrowright release}})\ holds\langle\underset{\cdot}{\underline{(V,1)}}\rangle\ busy\langle\underset{\cdot}{\underline{V}}\rangle \tag{42}$$

$$k(\underset{\cdot}{\underline{V \curvearrowright release}})\ holds\langle(V,\underset{N}{\underline{s(N)}})\rangle \tag{43}$$

$$k(\underset{\mathsf{throw\ new\ LockNotHeldEx}}{\underline{\quad V \curvearrowright release\quad}})\ holds(LTS) \Leftarrow V \notin LTS \tag{44}$$

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

Statement S ::= *all prior statements* | spawn E ; | acquire E ; | release E ;

Figure 12. Concurrent KOOL rules and added syntax

representing primitive values had to be coded in this fashion. The version of the code included in the appendix includes this implementation. In the version of KOOL soon to be released, this is changed to allow primitives to be invoked by number (similarly to Smalltalk) with a few named expressions. Special classes, identified with the keyword primclass instead of class, are allowed to hold primitive values. The advantage to this scheme is that users can define primitive classes themselves and can include these directly in KOOL source files. This also makes it easier to extend the standard classes in the language prelude without having to work at the level of the rewrite engine.

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 present a concurrent extension to the KOOL language; Concurrent KOOL is actually the language version we use as the basis for our other research.

To support concurrency, a new statement, spawn, was added to create new threads; threads are 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 can *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 needed 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 this is a significant new feature which makes significant changes to the state infrastructure, most of the rules are new – very few existing rules needed to be changed. Two rules, Rule 34 and Rule 35, are concurrent versions of Rule 1 and 2, respectively, and are changed by simply boxing them, which in K indicates that the rule can compete with other boxed rules. One rule, Rule (36), actually does change to take account of the new state infrastructure. This is the concurrent version of Rule 10. Rule 36 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 (37) 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 (38) 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 (39).

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 (40) and (41). In Rule (40), 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 (41), 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 (42), (43), and (44). In rule (42), 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* set. Rule (43) 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

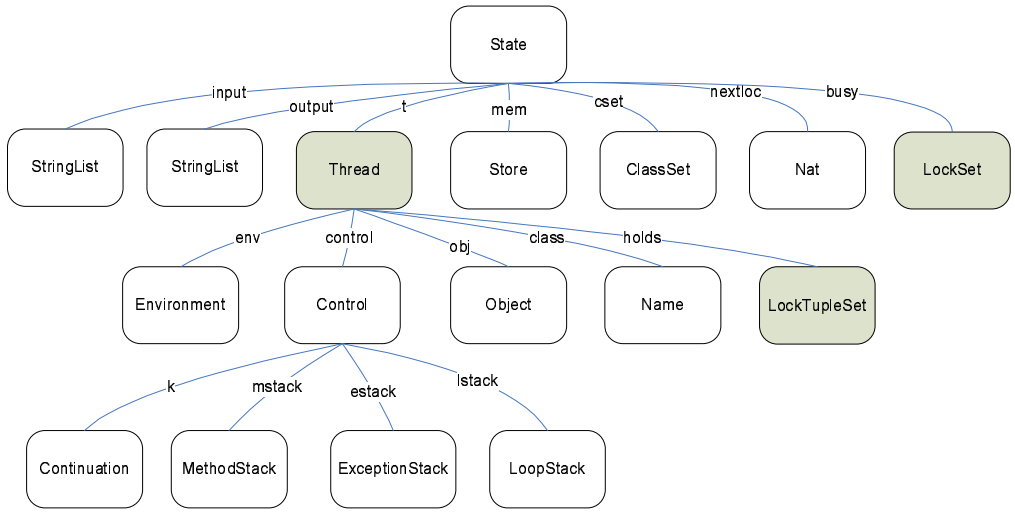


Figure 11. KOOL state infrastructure, with concurrency

```
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

be thrown. This is shown in Rule (44), 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 [15].

5. Conclusion and Future Work

In this paper we have presented the definition of KOOL, an experimental, object-oriented language with support for concurrency. While providing an interesting example of an OO language designed using the K framework, KOOL also is providing us a base upon which to experiment. We are currently looking into various type systems for KOOL, including domain-specific systems for security and units of measurement. We are also looking into other language features, including features related to modularity. The ability to quickly prototype language changes is to our advantage, since we can quickly modify the language and determine if the changes are useful or not.

In the broader realm of K definitions, we intend to implement a parser for K and a translator into rewriting logic in the near future. However, as explained in [18], 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 [7]. This would provide us with the ability to directly take the KOOL definition and generate an interpreter or compiler, providing for increased performance.

References

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

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

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706, pages 76–87. Springer LNCS, 2003.

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

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

A KOOL: SDF Grammar

KOOL is parsed using the SDF (Syntax Definition Formalism) notation and tools.

```
definition
module Threads
imports Stmt Exp
exports
  context-free syntax
    "spawn" Exp ";" -> Stmt { cons("Spawn") }
    "acquire" Exp ";" -> Stmt { cons("Acquire") }
    "release" Exp ";" -> Stmt { cons("Release") }
  lexical syntax
    "spawn" -> Name { reject }
    "acquire" -> Name { reject }
    "release" -> Name { reject }

module Comments
imports basic/Whitespace
exports
  sorts BlockComment CommentPart Asterisk Slash
  lexical syntax
    "//" ~[\n]* [\n] -> LAYOUT
    BlockComment -> LAYOUT
    "/*" CommentPart* "*/" -> BlockComment
    ~[\/\*]      -> CommentPart
    Asterisk     -> CommentPart
    Slash        -> CommentPart
    BlockComment -> CommentPart
    [\/] -> Slash
    [\*] -> Asterisk

  lexical restrictions
    Asterisk -/- [\/]
    Slash    -/- [\*]

module String
exports
  sorts String
  lexical syntax
   "\""  ~[\"\\\n\r]* "\"" -> String
  variables
    "Str"[0-9\']* -> String

module Char
exports
  sorts Char
  lexical syntax
    "'"[a-zA-Z0-9]"'" -> Char
  variables
    "Char"[0-9\']* -> Char

module Bool
imports Name
exports
```

```
  sorts Bool
  lexical syntax
    "true" -> Bool
    "false" -> Bool
  variables
    "Bool"[0-9\']* -> Bool
  lexical syntax
    "true" -> Name { reject }
    "false" -> Name { reject }

module TypeCase
imports Stmt Name Exp
exports
  sorts Case ElseCase
  context-free syntax
    "typecase" Exp "of" Case+  "end" -> Stmt { cons("TypeCase") }
    "typecase" Exp "of" Case+ ElseCase "end" -> Stmt { cons("TypeCaseElse") }
    "case" Name "of" Stmt -> Case { cons("Case") }
    "else" Stmt -> ElseCase { cons("ElseCase") }
  lexical syntax
    "typecase" -> Name { reject }
    "case" -> Name { reject }
    "else" -> Name { reject }
    "end" -> Name { reject }

module Exception
imports Stmt Exp
exports
  context-free syntax
    "try" Stmt "catch" Name Stmt "end" -> Stmt { cons("TryCatch") }
    "throw" Exp ";" -> Stmt { cons("Throw") }
  lexical syntax
    "try" -> Name { reject }
    "catch" -> Name { reject }
    "end" -> Name { reject }
    "throw" -> Name { reject }

module Number
exports
  sorts Integer Float
  lexical syntax
    [\+\-]?[0-9]+ -> Integer
    [\+\-]?"."[0-9]+ -> Float
    [\+\-]?[0-9]+"." -> Float
    [\+\-]?[0-9]+"."[0-9]+ -> Float
  variables
    "Int"[0-9\']* -> Integer
    "Float"[0-9\']* -> Float


module New
imports Exp Name
exports
  context-free syntax
```

```
    "new" Name -> Exp { cons("NewNoParams") }
    "new" Name "(" {Exp ","}* ")" -> Exp { cons("New") }
  lexical syntax
    "new" -> Name { reject }

module Send
imports Exp Stmt Name
exports
  context-free syntax
    Exp OperatorName Exp -> Exp {left, cons("BinOpSend") }
    Exp "." Name ("(" ")")? -> Exp { cons("UnarySend") }
    Exp "." Name "(" {Exp ","}+ ")" -> Exp { cons("Send") }
    "return" ";" -> Stmt { cons("ReturnNoVal") }
    "return" Exp ";" -> Stmt { cons("Return") }
  lexical syntax
    "return" -> Name { reject }
  context-free priorities
    {left:
      Exp "." Name ("(" ")")?         -> Exp
      Exp "." Name "(" {Exp ","}+ ")" -> Exp
    }
  > Exp OperatorName Exp              -> Exp



module Super
imports Name Exp Send
exports
  context-free syntax
    "super" "." Name "(" { Exp "," }+ ")" -> Exp { cons("Super") }
    "super" "." Name ( "("  ")" )? -> Exp { cons("SuperUnary") }
    "super" "(" ")" -> Exp { cons("SuperUnarySame") }
    "super" "(" { Exp "," }+ ")" -> Exp { cons("SuperSame") }
  lexical syntax
    "super" -> Name { reject }

module Self
imports Name Exp
exports
  context-free syntax
    "self" -> Exp { cons("Self") }
  lexical syntax
    "self" -> Name { reject }

module Loop
imports Stmt Exp Name
exports
  context-free syntax
    "for" Name "<-" Exp "to" Exp "do" Stmt "od" -> Stmt { cons("ForLoop") }
    "while" Exp "do" Stmt "od" -> Stmt { cons("WhileLoop") }
    "break" ";" -> Stmt { cons("Break") }
    "continue" ";" -> Stmt { cons("Continue") }
  lexical syntax
    "for" -> Name { reject }
```

```
    "to" -> Name { reject }
    "do" -> Name { reject }
    "od" -> Name { reject }
    "while" -> Name { reject }
    "break" -> Name { reject }
    "continue" -> Name { reject }

module Conditional
imports Stmt Exp
exports
  context-free syntax
    "if" Exp "then" Stmt "else" Stmt "fi" -> Stmt { cons("IfThenElse") }
    "if" Exp "then" Stmt "fi" -> Stmt { cons("IfThen") }
  lexical syntax
    "if" -> Name { reject }
    "then" -> Name { reject }
    "else" -> Name { reject }
    "fi" -> Name { reject }

module Assignment
imports Stmt Exp Name
exports
  context-free syntax
    Exp "<-" Exp ";" -> Stmt {cons("Assign")}
  lexical syntax
    "<-" -> Name { reject }
    "<-" -> OperatorName { reject }

module SeqComp
imports Stmt
exports
  context-free syntax
    Stmt Stmt -> Stmt {right, cons("SeqComp")}

module Block
imports Stmt Decl Name
exports
  context-free syntax
    "begin" Decl* Stmt "end" -> Stmt {cons("Block")}
  lexical syntax
    "begin" -> Name { reject }
    "end" -> Name { reject }

module Stmt
imports Exp
exports
  sorts Stmt
  context-free syntax
    Exp ";" -> Stmt { cons("StmtExp") }

module Method
imports Name Decl Stmt
exports
  sorts Method
```

```
  context-free syntax
    "method" Name "is" Decl* Stmt "end" -> Method { cons("MethodDef") }
    "method" Name "(" {Name ","}+ ")" "is" Decl* Stmt "end" -> Method { cons("MethodWParamsDef") }
  lexical syntax
    "method" -> Name { reject }
    "is" -> Name { reject }
    "end" -> Name { reject }

module Decl
imports Name
exports
  sorts Decl
  context-free syntax
    "var" {Name ","}+ ";" -> Decl { cons("Decl") }
  lexical syntax
    "var" -> Name { reject }

module Class
imports Decl Name Method
exports
  sorts Class
  context-free syntax
    "class" Name "is" Decl* Method* "end" -> Class { cons("ClassNoParent") }
    "class" Name "extends" Name "is" Decl* Method* "end" -> Class { cons("Class") }
  lexical syntax
    "class" -> Name { reject }
    "is" -> Name { reject }
    "end" -> Name { reject }
    "extends" -> Name { reject }

module basic/Whitespace

exports
  lexical syntax
    [\ \t\n\r] -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n\r]

module Name
imports basic/Whitespace
exports
  sorts Name OperatorName
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]* -> Name
    [\+\-\/\*\<\>\=\!\^\:\#\%]+ -> OperatorName
  variables
    "Id"[0-9\']* -> Name
  context-free syntax
    OperatorName -> Name { cons("OpName") }

module Exp
imports Name Number String Char Bool
exports
  sorts Exp
```

```
  context-free syntax
    Name -> Exp { cons("Name") }
    Integer -> Exp { cons("Integer") }
    Float -> Exp { cons("Float") }
    Bool -> Exp { cons("Bool") }
    Char -> Exp { cons("Char") }
    String -> Exp { cons("String") }
   "(" Exp ")" -> Exp { bracket }
   "nil" -> Exp { cons("Nil") }
  lexical syntax
    "nil" -> Name { reject }

module Program
imports Class Exp
exports
  sorts Program
  context-free start-symbols Program
  context-free syntax
    Class* Exp -> Program { cons("Program") }

module Main
imports Program Class Block SeqComp Assignment Conditional Loop
        Self Super Send New Number Exception TypeCase Bool Char
        String Comments Threads
```

B KOOL: Maude Syntax

```
fmod NAME-SYNTAX is
  protecting QID .
  sort Name Names .
  subsort Name < Names .

  op n : Qid -> Name .
  op empty : -> Names .
  op _,_ : Names Names -> Names [assoc id: empty] .
  ops Object Integer Float Boolean Char String Console console : -> Name .
  ops Lock LockManager : -> Name .
  ops Exception NilPointerException MethodNotFoundException : -> Name .
  ops InvalidSignatureException LockNotHeldException : -> Name .
  eq Object = n('Object) .
  eq Integer = n('Integer) .
  eq Float = n('Float) .
  eq Boolean = n('Boolean) .
  eq Char = n('Char) .
  eq String = n('String) .
  eq Console = n('Console) .
  eq console = n('console) .
  eq Exception = n('Exception) .
  eq NilPointerException = n('NilPointerException) .
  eq MethodNotFoundException = n('MethodNotFoundException) .
  eq InvalidSignatureException = n('InvalidSignatureException) .
  eq LockNotHeldException = n('LockNotHeldException) .
endfm

***
*** Scalars are just considered objects. However, since it is useful
*** to get at the value for constructing the object, and since in
*** the syntax we will just see this as the scalar (like 5, or "hello"),
*** we will treat these separately from the other names here.
***
fmod SCALARS-SYNTAX is
  including INT .
  including FLOAT .
  including BOOL .
  including STRING .

  sorts KInt KFloat KBool KChar KString .
  op i : Int -> KInt .
  op f : Float -> KFloat .
  op b : Bool -> KBool .
  op c : Char -> KChar .
  op s : String -> KString .
endfm

***
*** Expressions.
*** Introduce subsorting based on syntax definition.
***
fmod EXP-SYNTAX is
```

```
  including NAME-SYNTAX .
  including SCALARS-SYNTAX .
  sorts Exp Exps .
  subsort Exp < Exps .
  subsort Name KInt KFloat KBool KChar KString < Exp .
  subsort Names < Exps .

  op empty : -> Exps .
  op _,_ : Exps Exps -> Exps [assoc id: empty] .
  op Nil : -> Exp .
endfm

***
*** Statements.
*** We can make an expression a statement by terminating
*** it with a ; (so a + 5; is a valid, if useless, statement).
***
fmod STMT-SYNTAX is
  including EXP-SYNTAX .
  sort Stmt .
  op _; : Exp -> Stmt .
endfm

***
*** Declaration statements.
***
fmod DECL-SYNTAX is
  including NAME-SYNTAX .

  sort Decl Decls .
  subsort Decl < Decls .

  op var_ : Names -> Decl .

  op empty : -> Decls .
  op _,_ : Decls Decls -> Decls [assoc id: empty] .

endfm

***
*** Sequential composition.
*** Unfortunately we need to add a prec here
*** and the gathering to make normal use more legible.
*** These make sequencing right-associative and push the
*** precedence lower than other ops. When used by the code
*** generator, we will prefix everything anyway, so this
*** won't matter then.
***
fmod SEQUENCE-SYNTAX is
  including STMT-SYNTAX .

  op __ : Stmt Stmt -> Stmt [prec 70 gather(e E)] .
endfm
```

```
***
*** Statement block.
***
fmod BLOCK-SYNTAX is
  including STMT-SYNTAX .
  including DECL-SYNTAX .

  op begin__end : Decls Stmt -> Stmt .
endfm

***
*** Assignment statements.
***
fmod ASSIGNMENT-SYNTAX is
  including EXP-SYNTAX .
  including STMT-SYNTAX .

  op _<-_; : Exp Exp -> Stmt .
endfm

***
*** Conditionals.
***
fmod CONDITIONAL-SYNTAX is
  including EXP-SYNTAX .
  including STMT-SYNTAX .

  op if_then_else_fi : Exp Stmt Stmt -> Stmt .
  op if_then_fi : Exp Stmt -> Stmt .
endfm

***
*** Loops.
***
fmod LOOP-SYNTAX is
  including EXP-SYNTAX .
  including STMT-SYNTAX .
  including NAME-SYNTAX .

  op for_<-_to_do_od : Name Exp Exp Stmt -> Stmt .
  op while_do_od : Exp Stmt -> Stmt .
  op break`; : -> Stmt .
  op continue`; : -> Stmt .
endfm

***
*** Methods
***
fmod METHOD-SYNTAX is
  including NAME-SYNTAX .
  including DECL-SYNTAX .
  including STMT-SYNTAX .

  sorts Method Methods .
```

```
  subsort Method < Methods .

  op empty : -> Methods .
  op _,_ : Methods Methods -> Methods [assoc id: empty] .

  op method_`(_`)is__end : Name Names Decls Stmt -> Method .

  op return`; : -> Stmt .
  op return_; : Exp -> Stmt .
endfm

***
*** Classes
***
fmod CLASS-SYNTAX is
  including DECL-SYNTAX .
  including NAME-SYNTAX .
  including METHOD-SYNTAX .

  sorts Class Classes .
  subsort Class < Classes .

  op _,_ : Classes Classes -> Classes [assoc id: empty] .
  op empty : -> Classes .

  op class_extends_is__end : Name Name Decls Methods -> Class .
endfm

***
*** Self
***
fmod SELF-SYNTAX is
  including NAME-SYNTAX .
  including EXP-SYNTAX .

  op self : -> Exp .
endfm

***
*** Super
***
fmod SUPER-SYNTAX is
  including NAME-SYNTAX .
  including EXP-SYNTAX .

  op super._`(_`) : Name Exps -> Exp .
  op super`(_`) : Exps -> Exp .
endfm

***
*** Send
***
fmod SEND-SYNTAX is
  including EXP-SYNTAX .
```

```
  including NAME-SYNTAX .
  including STMT-SYNTAX .

  op _._`(_`) : Exp Name Exps -> Exp .
endfm

***
*** New
***
fmod NEW-SYNTAX is
  including EXP-SYNTAX .
  including NAME-SYNTAX .

  op new_`(_`) : Name Exps -> Exp .
endfm

***
*** Exceptions
***
fmod EXCEPTION-SYNTAX is
  including STMT-SYNTAX .
  including EXP-SYNTAX .

  op try_catch__end : Stmt Name Stmt -> Stmt .
  op throw_; : Exp -> Stmt .
endfm

***
*** Typecase
***
fmod TYPECASE-SYNTAX is
  including NAME-SYNTAX .
  including EXP-SYNTAX .
  including STMT-SYNTAX .

  sorts ElseCase Case Cases .
  subsort Case < Cases .

  op empty : -> Cases .
  op _,_ : Cases Cases -> Cases [assoc id: empty] .

  op case_of_ : Name Stmt -> Case .
  op typecase_of_end : Exp Cases -> Stmt .
  op typecase_of__end : Exp Cases ElseCase -> Stmt .
  op else_ : Stmt -> ElseCase .
endfm

***
*** Programs
***
fmod PROGRAM-SYNTAX is
  including CLASS-SYNTAX .
  including EXP-SYNTAX .
```

```
  sort Program .

  op __ : Classes Exp -> Program .
endfm

***
*** Concurrency operations
***
fmod CONCURRENCY-SYNTAX is
  including STMT-SYNTAX .
  including EXP-SYNTAX .

  op spawn_; : Exp -> Stmt .
  op acquire_; : Exp -> Stmt .
  op release_; : Exp -> Stmt .
endfm

***
*** Bring all syntax together here.
***
fmod MAIN-SYNTAX is
  including PROGRAM-SYNTAX .
  including CLASS-SYNTAX .
  including BLOCK-SYNTAX .
  including SEQUENCE-SYNTAX .
  including ASSIGNMENT-SYNTAX .
  including CONDITIONAL-SYNTAX .
  including LOOP-SYNTAX .
  including SELF-SYNTAX .
  including SUPER-SYNTAX .
  including SEND-SYNTAX .
  including NEW-SYNTAX .
  including SCALARS-SYNTAX .
  including EXCEPTION-SYNTAX .
  including TYPECASE-SYNTAX .
  including CONCURRENCY-SYNTAX .
endfm
```

C KOOL: Infrastructure

```
fmod CONTINUATION is
  sorts ContinuationItem Continuation .
  subsort ContinuationItem < Continuation .

  op empty : -> Continuation .
  op _->_ : Continuation Continuation -> Continuation [assoc id: empty] .
endfm

fmod VALUE is
  including CONTINUATION .
  sorts Value ValueList .
  subsort Value < ValueList .

  op empty : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: empty] .
  op val : ValueList -> Continuation .
  op nil : -> Value .
endfm

fmod LOCATION is
  including NAT .
  sorts Location LocationList .
  subsort Location < LocationList .

  op empty : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: empty] .
  op loc : Nat -> Location .
  op locs : Nat Nat -> LocationList .

  vars N M : Nat .

  eq locs(N, 0) = empty .
  eq locs(N, s(M)) = loc(N), locs(s(N), M) .
endfm

fmod ENVIRONMENT is
  including NAME-SYNTAX .
  including LOCATION .
  sort Env .

  op empty : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: empty] .
  op _[_] : Env Name -> Location .
  op _[_<-_] : Env Names LocationList -> Env .

  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : Names .  var Ll : LocationList .

  eq ([X,L] Env)[X] = L .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[empty <- empty] = Env .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
  eq Env[X <- L] = (Env [X,L]) [owise] .
endfm

fmod STORE is
  including LOCATION .
  including VALUE .
  sort Store .

  op empty : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: empty] .
  op _[_] : Store Location -> Value .
```

```
  op _[_<-_] : Store LocationList ValueList -> Store .
  op _[_<*-_] : Store LocationList Value -> Store .

  var L : Location .  var Mem : Store .  vars V V' : Value .
  var Ll : LocationList .  var Vl : ValueList .

  eq ([L,V] Mem)[L] = V .
  eq Mem[empty <- empty] = Mem .
  eq ([L,V] Mem)[L,Ll <- V',Vl] = ([L,V'] Mem)[Ll <- Vl] .
  eq Mem[L,Ll <- V',Vl] = (Mem [L,V'])[Ll <- Vl] [owise] .

  eq Mem[empty <*- V] = Mem .
  eq ([L,V] Mem)[L,Ll <*- V'] = ([L,V'] Mem)[Ll <*- V'] .
  eq Mem[L,Ll <*- V'] = (Mem [L,V'])[Ll <*- V'] [owise] .

endfm


***
*** Internal definition of a method
***
fmod METHOD is
  including METHOD-SYNTAX .
  including CONTINUATION .

  sorts MethodItem IMethod MethodSet .
  subsort IMethod < MethodSet .

  op empty : -> MethodSet .
  op __ : MethodSet MethodSet -> MethodSet [assoc comm id: empty] .

  op mthd : MethodItem -> IMethod .

  op empty : -> MethodItem .
  op __ : MethodItem MethodItem -> MethodItem [assoc comm id: empty] .

  op mname : Name -> MethodItem .
  op mparams : Names -> MethodItem .
  op mdecls : Names -> MethodItem .
  op mbody : Continuation -> MethodItem .
endfm

***
*** Internal definition of a class
***
fmod CLASS is
  including METHOD .
  including CLASS-SYNTAX .

  sorts ClassItem IClass ClassSet .
  subsort IClass < ClassSet .

  op empty : -> ClassSet .
  op __ : ClassSet ClassSet -> ClassSet [assoc comm id: empty] .

  op cls : ClassItem -> IClass [format (b! o)] .

  op empty : -> ClassItem .
  op __ : ClassItem ClassItem -> ClassItem [assoc comm id: empty] .

  sort NameSet .
  subsort Name < NameSet .
  op emptyNS : -> NameSet .
  op __ : NameSet NameSet -> NameSet [assoc comm id: emptyNS] .

  op cname : Name -> ClassItem .
  op pname : Name -> ClassItem .
  op mthds : MethodSet -> ClassItem .
  op flds  : Names -> ClassItem .
```

```
  op inheritsSet : NameSet -> ClassItem .
endfm

***
*** The object environment holds the environment
*** information for each "layer" of the object. The
*** layer is keyed on a nat to line up with the
*** class IDs that are used in the class definition
*** above -- this allows us to tie each layer to
*** a specific class.
***
fmod OBJ-ENVIRONMENT is
  including ENVIRONMENT .
  sort ObjEnv .
  op empty : -> ObjEnv .
  op [_,_] : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: empty] .
endfm

***
*** An object is the instantiation of a specific class,
*** and contains the information from that class and all
*** superclasses. To make the object extensible, we will
*** represent it as a "soup" of information -- this will
*** be especially useful later, if we decide to add more
*** features to our language.
***
fmod OBJECT is
  including VALUE .
  including CLASS .
  including ENVIRONMENT .
  including OBJ-ENVIRONMENT .

  sort Object .

  op empty : -> Object .
  op __ : Object Object -> Object [assoc comm id: empty] .

  op oenv : ObjEnv -> Object .
  op myclass : Name -> Object .

  op o : Object -> Value .
endfm

fmod STRING-LIST is
  including STRING .
  sort StringList .
  subsort String < StringList .

  op empty : -> StringList .
  op _,_ : StringList StringList -> StringList [assoc id: empty] .
endfm

fmod LOCK is
  including INT .
  including VALUE .

  sorts Lock LockSet LockTuple LockTupleSet .
  subsort Lock < LockSet .
  subsort LockTuple < LockTupleSet .

  op empty : -> LockSet .
  op __ : LockSet LockSet -> LockSet [assoc comm id: empty] .
  op empty : -> LockTupleSet .
  op __ : LockTupleSet LockTupleSet -> LockTupleSet [assoc comm id: empty] .
  op lk : Value -> Lock .
  op [_,_] : Lock Nat -> LockTuple .
  op notin : LockSet Lock -> Bool .
  op notin : LockTupleSet Lock -> Bool .
```

```
  op _-_ : LockSet LockTupleSet -> LockSet .

  var LK : Lock . var LS : LockSet . var LTS : LockTupleSet . var I : Int .
  var N : Nat .

  eq notin(LS LK, LK) = false .
  eq notin(LS,LK) = true [owise] .
  eq notin(LTS [LK,N], LK) = false .
  eq notin(LTS, LK) = true [owise] .
  eq (LK LS) - ([LK,N] LTS) = LS - LTS .
  eq LS - empty = LS .
endfm

*******************************************************************************
***
*** Program state "soup".
***
*******************************************************************************
fmod STATE is
  including CONTINUATION .
  including VALUE .
  including LOCATION .
  including ENVIRONMENT .
  including STORE .
  including METHOD .
  including CLASS .
  including OBJ-ENVIRONMENT .
  including OBJECT .
  including STRING-LIST .
  including LOCK .

  sorts State StateList .
  subsort State < StateList .

  op empty : -> State .
  op __ : State State -> State [assoc comm id: empty] .

  op empty : -> StateList .
  op _,_ : StateList StateList -> StateList [assoc id: empty] .

  sorts MStackTuple MStackTupleList .
  subsort MStackTuple < MStackTupleList .
  op empty : -> MStackTupleList .
  op _,_ : MStackTupleList MStackTupleList -> MStackTupleList [assoc id: empty] .
  op [_,_,_,_,_] : Continuation State Env Object Name -> MStackTuple .

  sorts EStackTuple EStackTupleList .
  subsort EStackTuple < EStackTupleList .
  op empty : -> EStackTupleList .
  op _,_ : EStackTupleList EStackTupleList -> EStackTupleList [assoc id: empty] .
  op [_,_,_,_,_,_] : Continuation State Env Object Name Continuation -> EStackTuple .

  sorts LStackTuple LStackTupleList .
  subsort LStackTuple < LStackTupleList .
  op empty : -> LStackTupleList .
  op _,_ : LStackTupleList LStackTupleList -> LStackTupleList [assoc id: empty] .
  op [_,_,_,_] : Continuation State Env Continuation -> LStackTuple .

  op control : State -> State [format (b! o)] .
  op k : Continuation -> State [format (y! o)] .
  op mstack : MStackTupleList -> State [format (mu! o)] .
  op estack : EStackTupleList  -> State [format (mu! o)] .
  op lstack : LStackTupleList -> State [format (mu! o)] .

  op env : Env -> State [format (r! o)] .
  op cobj : Object -> State [format (r! o)] .
  op cclass : Name -> State [format (r! o)] .
  op cset : ClassSet -> State [format (r! o)] .
  op mem : Store -> State [format (r! o)] .
```

```
  op nextLoc : Nat -> State [format (r! o)] .
  op nextoid : Nat -> State [format (r! o)] .
  op input : StringList -> State [format (r! o)] .
  op output : StringList -> State [format (r! o)] .
  op holds : LockTupleSet -> State [format (r! o)] .
  op busy : LockSet -> State [format (r! o)] .

  op t : State -> State [format (r! o)] .

***
*** Placeholder for the starter environment
***
  op baseenv : -> Env .

endfm

*******************************************************************************
***
*** Helper modules
***
*** These modules provide helpers that are used to manipulate the
*** state more easily, for instance to allocate a number of names
*** at once.
***
*******************************************************************************
mod STATE-HELPERS is
  including STATE .

  op noval : -> Value .

  vars X Xc Xc' : Name . var Xl : Names . vars K K' : Continuation . vars N N' : Nat .
  vars CS CS' : State . vars V V' : Value . var Vl : ValueList . vars Env Env' : Env .
  var Mem : Store . var L : Location . vars SL SL' : StateList .
  vars O O' : Object . vars C C' : IClass . var Ll : LocationList .
  var CI : ClassItem . var Cs : ClassSet . var OE : ObjEnv .
  var TS TS' : State .

  op len : Names -> Nat .
  eq len(X,Xl) = 1 + len(Xl) .
  eq len((empty).Names) = 0 .

  op len : ValueList -> Nat .
  eq len(V,Vl) = 1 + len(Vl) .
  eq len((empty).ValueList) = 0 .

  op saveenv : Continuation -> Continuation .
  eq control(k(saveenv(K') -> K) CS) env(Env) =
     control(k(K' -> wrapenv(Env) -> K) CS) env(Env) .
  eq k(val(Vl) -> saveenv(K') -> K) = k(saveenv(val(Vl) -> K') -> K) .

  op wrapenv : Env -> Continuation .
  eq control(k(wrapenv(Env) -> K) CS) env(Env') =
     control(k(K) CS) env(Env) .
  eq k(val(Vl) -> wrapenv(Env) -> K) = k(wrapenv(Env) -> val(Vl) -> K) .

  op bind : Names -> Continuation .
 ceq t(control(k(val(Vl) -> bind(Xl) -> K) CS) env(Env) TS) mem(Mem) nextLoc(N) =
     t(control(k(K) CS) env(Env[Xl <- Ll]) TS) mem(Mem[Ll <- Vl]) nextLoc(N + N')
  if Ll := locs(N,len(Xl)) /\ N' := len(Xl) .

 ceq t(control(k(bind(Xl) -> K) CS) env(Env) TS) mem(Mem) nextLoc(N) =
     t(control(k(K) CS)  env(Env[Xl <- Ll]) TS) mem(Mem[Ll <*- nil]) nextLoc(N + N')
  if Ll := locs(N,len(Xl)) /\ N' := len(Xl) .

  op assignTo_ : Name -> Continuation .
  eq control(k(val(V) -> assignTo(X) -> K) CS) env([X,L] Env) cclass(Xc) =
     control(k(val(V) -> lassign(L) -> K) CS) env([X,L] Env)  cclass(Xc) .
  eq control(k(val(V) -> assignTo(X) -> K) CS) env(Env) cclass(Xc) =
     control(k(val(V) -> oassignTo(X,Xc) -> K) CS) env(Env) cclass(Xc) [owise] .
```

```
  op oassignTo : Name Name -> Continuation .
  eq control(k(val(V) -> oassignTo(X,Xc) -> K) CS) cobj(O oenv([Xc,[X,L] Env] OE)) =
     control(k(val(V) -> lassign(L) -> K) CS) cobj(O oenv([Xc,[X,L] Env] OE)) .
  eq t(control(k(val(V) -> oassignTo(X,Xc) -> K) CS) cobj(O) TS) cset(Cs cls(cname(Xc) pname(Xc') CI)) =
     t(control(k(val(V) -> oassignTo(X,Xc') -> K) CS) cobj(O) TS) cset(Cs cls(cname(Xc) pname(Xc') CI)) [owise] .

  op lassign : Location -> Continuation .
  rl t(control(k(val(V) -> lassign(L) -> K) CS) TS) mem(Mem [L,V']) =>
     t(control(k(K) CS) TS) mem(Mem [L,V]) .

  op lookup_ : Name -> Continuation .
  eq control(k(lookup(X) -> K) CS) env([X,L] Env) cclass(Xc) =
     control(k(llookup(L) -> K) CS) env([X,L] Env) cclass(Xc) .
  eq control(k(lookup(X) -> K) CS) env(Env) cclass(Xc) =
     control(k(olookup(X,Xc) -> K) CS) env(Env) cclass(Xc) [owise] .

  op olookup : Name Name -> Continuation .
  eq control(k(olookup(X,Xc) -> K) CS) cobj(O oenv([Xc,[X,L] Env] OE)) =
     control(k(llookup(L) -> K) CS) cobj(O oenv([Xc,[X,L] Env] OE)) .
  eq t(control(k(olookup(X,Xc) -> K) CS) cobj(O) TS) cset(Cs cls(cname(Xc) pname(Xc') CI)) =
     t(control(k(olookup(X,Xc') -> K) CS) cobj(O) TS) cset(Cs cls(cname(Xc) pname(Xc') CI)) [owise] .

  op llookup : Location -> Continuation .
  rl t(control(k(llookup(L) -> K) CS) TS) mem(Mem [L,V]) =>
     t(control(k(val(V) -> K) CS) TS) mem(Mem [L,V]) .

  vars MSTL MSTL' : MStackTupleList .
  op pushMStack : Continuation State Env Object Name -> Continuation .
  eq control(k(pushMStack(K,CS mstack(MSTL'),Env,O,Xc) -> K') mstack(MSTL) CS') =
     control(k(K') mstack([K,CS,Env,O,Xc], MSTL) CS') .

  op popMStack : -> Continuation .
  eq control(k(popMStack -> K) mstack([K',CS',Env',O,Xc],MSTL) CS)
       env(Env) cobj(O') cclass(Xc') =
     control(k(K') mstack(MSTL) CS') env(Env') cobj(O) cclass(Xc) .
  eq control(k(val(Vl) -> popMStack -> K) mstack([K',CS',Env',O,Xc],MSTL) CS)
       env(Env) cobj(O') cclass(Xc') =
     control(k(val(Vl) -> K') mstack(MSTL) CS') env(Env') cobj(O) cclass(Xc) .

  vars ESTL ESTL' : EStackTupleList . var Kc : Continuation .
  op pushEStack : Continuation State Env Object Name Continuation -> Continuation .
  eq control(k(pushEStack(K,CS estack(ESTL'),Env,O,Xc,Kc) -> K') estack(ESTL) CS') =
     control(k(K') estack([K,CS,Env,O,Xc,Kc], ESTL) CS') .

***
*** Just discard the expression stack item, getting us back to prior
*** state. Igore Kc, since this only happens when we don't have
*** an exception thrown.
***
  op popEStack : -> Continuation .
  eq control(k(popEStack -> K) estack([K',CS',Env',O,Xc,Kc],ESTL) CS)
       env(Env) cobj(O') cclass(Xc') =
     control(k(K') estack(ESTL) CS') env(Env') cobj(O) cclass(Xc) .

***
*** Handle the loop stack. The first operation will push the needed recovery
*** context onto the loop stack. The second will pop the context off and replace
*** the current context with what has been saved. The third will put in the
*** "continue" continuation but will leave the stack alone, since we are
*** still executing the loop.
***
  vars LSTL LSTL' : LStackTupleList .
  op pushLStack : Continuation State Env Continuation -> ContinuationItem .
  eq control(k(pushLStack(K,CS lstack(LSTL'),Env,Kc) -> K') lstack(LSTL) CS') =
     control(k(K') lstack([K,CS,Env,Kc], LSTL) CS') .
  op popLStack : -> Continuation .
  eq control(k(popLStack -> K) lstack([K',CS',Env',Kc],LSTL) CS) env(Env) =
     control(k(K') lstack(LSTL) CS') env(Env') .
```

```
  op contLStack : -> ContinuationItem .
  eq control(k(contLStack -> K) lstack([K',CS',Env',Kc],LSTL) CS) env(Env) =
     control(k(Kc) lstack([K',CS',Env',Kc],LSTL) CS') env(Env') .

  op clearLStack : -> ContinuationItem .
  eq control(k(clearLStack -> K) CS lstack(LSTL)) =
     control(k(K) CS lstack(empty)) .
***
*** This is designed specifically for when we want to run the catch code stored
*** in Kc. We will run that, and the remaining continuation, in the same
*** environment (although we may always throw again). The Vl should be the
*** thrown object.
***
  op popAndRunEStack : -> Continuation .
  eq control(k(val(Vl) -> popAndRunEStack -> K) estack([K',CS',Env',O,Xc,Kc],ESTL) CS)
       env(Env) cobj(O') cclass(Xc') =
     control(k(val(Vl) -> Kc -> K') estack(ESTL) CS') env(Env') cobj(O) cclass(Xc) .

  op resetObjectTo : Object -> Continuation .
  eq control(k(resetObjectTo(O) -> K) CS) cobj(O') =
     control(k(K) CS) cobj(O) .
  eq k(val(Vl) -> resetObjectTo(O) -> K) = k(resetObjectTo(O) -> val(Vl) -> K) .

  op resetClassTo : Name -> Continuation .
  eq control(k(resetClassTo(Xc) -> K) CS) cclass(Xc') =
     control(k(K) CS) cclass(Xc) .
  eq k(val(Vl) -> resetClassTo(Xc) -> K) = k(resetClassTo(Xc) -> val(Vl) -> K) .

  op resetClassAndObjectTo : Name Object -> Continuation .
  eq control(k(resetClassAndObjectTo(Xc,O) -> K) CS) cclass(Xc') cobj(O') =
     control(k(K) CS) cclass(Xc) cobj(O) .
  eq k(val(Vl) -> resetClassAndObjectTo(Xc,O) -> K) = k(resetClassAndObjectTo(Xc,O) -> val(Vl) -> K) .

  op getCurrentObject : -> ContinuationItem .
  eq control(k(getCurrentObject -> K) CS) cobj(O) =
     control(k(val(o(O)) -> K) CS) cobj(O) .

***
*** Value discard
***
 op discard : -> ContinuationItem .
 eq k(val(V) -> discard -> K) = k(K) .

endm
```

D KOOL: Dynamic Semantics

```
***
*** Semantics for expressions. This handles general processing of
*** expression lists. This is more complex than it needs to be
*** since we do not always evaluate at the top of the continuation.
***
mod EXP-SEMANTICS is
  including STATE .
  including STATE-HELPERS .
  including EXP-SYNTAX .

  op exp : Exps -> ContinuationItem .

  var E : Exp . var El : Exps .
  var V : Value . vars Vl Vl' : ValueList .
  var K : Continuation . vars Cs Cs' : ContinuationList .

***
*** So we can maintain parallelism in evaluation, put the
*** expression list into a continuation list structure that
*** will allow us to collect values in the correct order.
***
  sort ContinuationList .
  subsort Continuation < ContinuationList .

  op empty : -> ContinuationList .
  op _,_ : ContinuationList ContinuationList -> ContinuationList [assoc id: empty] .
  op clist : ContinuationList -> Continuation .
  op {_|_} : ContinuationList ContinuationList -> Continuation .
  op kexp : Exp -> Continuation .

***
*** An empty expression list always evaluates to an empty value list. This can
*** happen on constructor invocations, method invocations, etc, where we have
*** no parameters.
***
 eq k(exp(empty) -> K) = k(val(empty) -> K) .

*** First, if we have an expression list, put it into
*** the continuation list
 ceq k(exp(E,El) -> K) = k(clist(kexp(E),kexp(El)) -> K) if El =/= empty .

***
*** Now, stretch the expression list out to the write,
*** turning terms into individual expression continuations
*** as we go.
***
 ceq clist(Cs,kexp(E,El),Cs') = clist(Cs,kexp(E),kexp(El),Cs') if El =/= empty .

***
*** Collapse any internal value continuations into single continuations
***
  eq clist(Cs,val(Vl),val(Vl'),Cs') = clist(Cs,val(Vl,Vl'),Cs') .

***
*** If we just have a value list, get rid of the wrapper
***
  eq clist(val(Vl)) = val(Vl) .

***
*** Finally, take out any expressions and evaluate them -- make
*** sure we only do this one at a time, and make sure we do this
*** in order (for things like method parameters, for instance,
*** we want a left to right evaluation order)
***
 eq clist(kexp(E),Cs) = exp(E) -> {empty | Cs} .
```

```
 eq clist(val(Vl),kexp(E),Cs) = exp(E) -> {val(Vl) | Cs} .
 eq val(V) -> {val(Vl) | Cs} = clist(val(Vl,V),Cs) .
 eq val(V) -> {empty | Cs} = clist(val(V),Cs) .

***
*** In some cases we still need the rule collapsing back lists
*** of values, so include it here.
***
 eq val(Vl) -> val(Vl') = val(Vl,Vl') .

***
*** Evaluate the Nil expression to the nil value
***
 eq exp(Nil) = val(nil) .
endm

mod STMT-SEMANTICS is
  including STATE .
  including STATE-HELPERS .
  including STMT-SYNTAX .
  including EXP-SEMANTICS .

  op stmt : Stmt -> ContinuationItem .

  var E : Exp .

***
*** To handle an expression statement,
*** evaluate the expression and discard
*** the results.
***
  eq stmt(E ;) = exp(E) -> discard .
endm

mod DECL-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including DECL-SYNTAX .

  var Ds : Decls . var Xs : Names .

  op decls : Decls -> Names .
  eq decls(var Xs, Ds) = Xs, decls(Ds) .
  eq decls(empty) = empty .
endm

mod NAME-SEMANTICS is
  including STATE-HELPERS .
  including NAME-SYNTAX .
  including EXP-SEMANTICS .

  var X : Name .

***
*** A name evaluates to whatever value
*** it has been assigned in the store.
***
  eq exp(X) = lookup(X) .
endm

mod SEND-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SEND-SYNTAX .
  including EXCEPTION-SYNTAX .
  including NEW-SYNTAX .

  vars Xm Xc Xc' : Name . vars K Km : Continuation .
```

```
  vars Xs Xs' : Names .
  var E : Exp . var Es : Exps . var Vl : ValueList .
  var CS : State . var CI : ClassItem . var Cs : ClassSet .
  vars O O' : Object . var Env : Env . var IC : IClass .
  var Ms : MethodSet . var TS : State .

***
*** First, evaluate the call target (which should evaluate to an
*** object) and the parameters (which can be empty).
  op toInvoke : Name -> ContinuationItem .
  op toInvokeAndWrap : Name -> ContinuationItem .
  eq exp(E . Xm ( Es )) = exp(E,Es) -> toInvoke(Xm) .

***
*** Second, now that we have an object back that we can invoke on,
*** save the current context and start searching for the method
*** in the dynamic class of the object.
*** Note that we clear the loop stack so we cannot break or continue
*** out of a method called inside a loop.
***
  op invoke : Name ValueList -> ContinuationItem .
  eq t(control(k(val(o(myclass(Xc) O), Vl) -> toInvoke(Xm) -> K) CS) cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc) CI) Cs) =
     t(control(k(pushMStack(K,CS,Env,O',Xc') -> clearLStack -> invoke(Xm,Vl)) CS) cclass(Xc) cobj(myclass(Xc) O)
       env(baseenv) TS) cset(cls(cname(Xc) CI) Cs) .

  eq t(control(k(val(o(myclass(Xc) O), Vl) -> toInvokeAndWrap(Xm) -> K) CS) cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc) CI) Cs) =
     t(control(k(pushMStack(discard -> val(o(myclass(Xc) O)) -> K,CS,Env,O',Xc') -> clearLStack -> invoke(Xm,Vl)) CS)
       cclass(Xc) cobj(myclass(Xc) O) env(baseenv) TS)
       cset(cls(cname(Xc) CI) Cs) .

***
*** If we invoke on a nil reference (like var x ; x + 5) throw an exception
***
  eq k(val(nil, Vl) -> toInvoke(Xm) -> K) =
     k(stmt(throw (new NilPointerException(empty)) ;) -> K) .

  eq k(val(nil, Vl) -> toInvokeAndWrap(Xm) -> K) =
     k(stmt(throw (new NilPointerException(empty)) ;) -> K) .

***
*** If we find the method to invoke, bind the formals and any declarations and
*** invoke the method body.
***
 ceq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS)
       cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Xs) mdecls(Xs') mbody(Km)) Ms) CI) Cs) =
     t(control(k(val(Vl) -> bind(Xs) -> bind(Xs') -> Km -> K) CS) cclass(Xc) TS)
       cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Xs) mdecls(Xs') mbody(Km)) Ms) CI) Cs)
  if len(Vl) == len(Xs) .
 ceq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS)
       cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Xs) mdecls(Xs') mbody(Km)) Ms) CI) Cs) =
     t(control(k(stmt(throw (new InvalidSignatureException(empty)) ;) -> K) CS) cclass(Xc) TS)
       cset(cls(cname(Xc) mthds(mthd(mname(Xm) mparams(Xs) mdecls(Xs') mbody(Km)) Ms) CI) Cs)
  if len(Vl) =/= len(Xs) .

***
*** If we don't find the method to invoke in the current class, switch context to the parent
*** class and keep looking. Change the name to match the parent class name when the method
*** and class have the same name (we are calling a constructor).
***
  eq t(control(k(invoke(Xm,Vl) -> K) CS) cclass(Xc) TS) cset(cls(cname(Xc) pname(Xc') CI) Cs) =
     t(control(k(invoke(if Xm == Xc then Xc' else Xm fi,Vl) -> K) CS) cclass(Xc') TS)
     cset(cls(cname(Xc) pname(Xc') CI) Cs) [owise] .

***
*** If we don't find the method at all, throw a MethodNotFound exception.
***
  var Q : Qid .
```

```
  eq t(control(k(invoke(n(Q),Vl) -> K) CS) cclass(n('Object)) TS) cset(Cs) =
     t(control(k(stmt(throw (new MethodNotFoundException(s(string(Q)))) ;) -> K) CS)
     cclass(n('Object)) TS) cset(Cs) [owise] .
endm

mod NEW-SEMANTICS is
  including STATE-HELPERS .
  including NEW-SYNTAX .
  including EXP-SEMANTICS .
  including SEND-SEMANTICS .

  vars X X' : Name . var Xs : Names . var CS : State .
  var K : Continuation . var CI : ClassItem . var Cs : ClassSet .
  var O : Object . var OE : ObjEnv . var Env : Env .
  var Es : Exps . var Vl : ValueList . var TS : State .

***
*** When we find a new, save the current environment and create the object
***
  op toCreate : Name -> ContinuationItem .
  eq k(exp(new X (Es) ) -> K) = k(exp(Es) -> toCreate(X) -> K) .
  eq val(Vl) -> toCreate(X) = saveenv(createObject(X) -> val(Vl) -> toInvokeAndWrap(X)) .


***
*** To create an object, save the current level and create the parent recursively.
*** If we hit object, just add the layer and stop recursing.
***
  op createObject : Name -> ContinuationItem .
  eq t(control(k(createObject(X) -> K) CS) TS) cset(cls(cname(X) pname(X') flds(Xs) CI) Cs) =
     t(control(k(createObject(X') -> addLayerFor(X,Xs) -> K) CS) TS) cset(cls(cname(X) pname(X') flds(Xs) CI) Cs) .
  eq k(createObject(n('Object)) -> K) = k(val(o(myclass(Object) oenv([Object,empty]))) -> K) .

***
*** To add a layer, first we need to bind the names in the layer we
*** are adding. Clear out the environment first, since we want to
*** save what we put into it next.
***
  op addLayerFor : Names -> ContinuationItem .
  eq control(k(val(o(myclass(X') O)) -> addLayerFor(X,Xs) -> K) CS) env(Env) =
     control(k(bind(Xs) -> bindObject(X,o(myclass(X) O)) -> K) CS) env(empty) .

***
*** Now we've added any bindings, so take the environment out
*** and save it as a layer in the object.
***
  op bindObject : Name Value -> ContinuationItem .
  eq control(k(bindObject(X,o(oenv(OE) O)) -> K) CS) env(Env) =
     control(k(val(o(oenv(OE [X,Env]) O)) -> K) CS) env(empty) .
endm

mod ASSIGNMENT-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including ASSIGNMENT-SYNTAX .

  var E  : Exp . var X : Name .

  eq stmt(X <- E ;) = exp(E) -> assignTo(X) .
endm

mod SEQUENCE-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SEQUENCE-SYNTAX .

  var S S' : Stmt .
```

```
  eq stmt(S S') = stmt(S) -> stmt(S') .

endm

mod BLOCK-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including BLOCK-SYNTAX .
  including DECL-SEMANTICS .

  var Ds : Decls . var S : Stmt . var Xs : Names .

  op block : -> ContinuationItem .

***
*** For blocks, first save the current env and then
*** process the decls.
***
  eq stmt(begin Ds S end) = saveenv(bind(decls(Ds)) -> stmt(S)) .
endm

mod PRIMITIVES-SEMANTICS is
  including INT .
  including RAT .
  including FLOAT .
  including BOOL .
  including STRING .
  including STATE-HELPERS .
  including NEW-SEMANTICS .
  including CONVERSION .
  including STRING-LIST .
  including EXCEPTION-SYNTAX .

  vars I I' : Int . vars F F' : Float . vars C C' : Char .
  vars B B' : Bool . vars S S' : String . var L : Location .
  var O : Object . vars V V' : Value . var OE : ObjEnv .
  var K : Continuation . vars TS CS : State . var Sl : StringList .
  var N : Nat .

  op pval : -> Name .

***
*** Generic ops, to pull out primitive components from objects
***
  op fetchPrimUnary : -> ContinuationItem .
  eq k(val(o(myclass(n('Integer)) O)) -> fetchPrimUnary -> K) =
     k(val(o(myclass(n('Integer)) O)) -> loadPrimInt -> K) .
  eq k(val(o(myclass(n('String)) O)) -> fetchPrimUnary -> K) =
     k(val(o(myclass(n('String)) O)) -> loadPrimString -> K) .
  eq k(val(o(myclass(n('Float)) O)) -> fetchPrimUnary -> K) =
     k(val(o(myclass(n('Float)) O)) -> loadPrimFloat -> K) .
  eq k(val(o(myclass(n('Boolean)) O)) -> fetchPrimUnary -> K) =
     k(val(o(myclass(n('Boolean)) O)) -> loadPrimBool -> K) .
  eq k(val(o(myclass(n('Char)) O)) -> fetchPrimUnary -> K) =
     k(val(o(myclass(n('Char)) O)) -> loadPrimChar -> K) .

  op fetchPrimBinary : -> ContinuationItem .
  op fetchPrimBinary : Value -> ContinuationItem .
  eq k(val(V,V') -> fetchPrimBinary -> K) =
     k(val(V') -> fetchPrimUnary -> fetchPrimBinary(V) -> K) .
  eq k(val(V') -> fetchPrimBinary(V) -> K) =
     k(val(V) -> fetchPrimUnary -> val(V') -> K) .
***
*** Memory ops, to handle primitive integer values
***
  op primInt : Int -> Value .

  op loadPrimInt : -> ContinuationItem .
```

```
  eq k(val(o(myclass(n('Integer)) oenv([n('Integer),[pval,L]] OE))) -> loadPrimInt -> K) =
     k(llookup(L) -> K) .

  op storePrimInt : Value -> ContinuationItem .
  eq k(val(o(myclass(n('Integer)) oenv([n('Integer),[pval,L]] OE))) -> storePrimInt(primInt(I)) -> K) =
     k(val(primInt(I)) -> lassign(L) -> val(o(myclass(n('Integer)) oenv([n('Integer),[pval,L]] OE))) -> K) .

***
*** Memory ops, to handle primitive string values
***
  op primString : String -> Value .

  op loadPrimString : -> ContinuationItem .
  eq k(val(o(myclass(n('String)) oenv([n('String),[pval,L]] OE))) -> loadPrimString -> K) =
     k(llookup(L) -> K) .

  op storePrimString : Value -> ContinuationItem .
  eq k(val(o(myclass(n('String)) oenv([n('String),[pval,L]] OE))) -> storePrimString(primString(S)) -> K) =
     k(val(primString(S)) -> lassign(L) -> val(o(myclass(n('String)) oenv([n('String),[pval,L]] OE))) -> K) .

***
*** Memory ops, to handle primitive float values
***
  op primFloat : Float -> Value .

  op loadPrimFloat : -> ContinuationItem .
  eq k(val(o(myclass(n('Float)) oenv([n('Float),[pval,L]] OE))) -> loadPrimFloat -> K) =
     k(llookup(L) -> K) .

  op storePrimFloat : Value -> ContinuationItem .
  eq k(val(o(myclass(n('Float)) oenv([n('Float),[pval,L]] OE))) -> storePrimFloat(primFloat(F)) -> K) =
     k(val(primFloat(F)) -> lassign(L) -> val(o(myclass(n('Float)) oenv([n('Float),[pval,L]] OE))) -> K) .

***
*** Memory ops, to handle primitive bool values
***
  op primBool : Bool -> Value .

  op loadPrimBool : -> ContinuationItem .
  eq k(val(o(myclass(n('Boolean)) oenv([n('Boolean),[pval,L]] OE))) -> loadPrimBool -> K) =
     k(llookup(L) -> K) .

  op storePrimBool : Value -> ContinuationItem .
  eq k(val(o(myclass(n('Boolean)) oenv([n('Boolean),[pval,L]] OE))) -> storePrimBool(primBool(B)) -> K) =
     k(val(primBool(B)) -> lassign(L) -> val(o(myclass(n('Boolean)) oenv([n('Boolean),[pval,L]] OE))) -> K) .

***
*** Memory ops, to handle primitive char values
***
  op primChar : Char -> Value .

  op loadPrimChar : -> ContinuationItem .
  eq k(val(o(myclass(n('Char)) oenv([n('Char),[pval,L]] OE))) -> loadPrimChar -> K) =
     k(llookup(L) -> K) .

  op storePrimChar : Value -> ContinuationItem .
  eq k(val(o(myclass(n('Char)) oenv([n('Char),[pval,L]] OE))) -> storePrimChar(primChar(C)) -> K) =
     k(val(primChar(C)) -> lassign(L) -> val(o(myclass(n('Char)) oenv([n('Char),[pval,L]] OE))) -> K) .

***
*** Primitive arithmetic and relational ops for ints, including
*** coersions to floats.
***
  ops primIntB+ primIntB- primIntB* primIntB/ primIntB% : -> ContinuationItem .
  ops primInt> primInt>= primInt< primInt<= primInt= primInt!= : -> ContinuationItem .

  eq k(val(primInt(I),primInt(I')) -> primIntB+ -> K) =
     k(exp(new Integer(empty)) -> storePrimInt(primInt(I + I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB- -> K) =
```

```
     k(exp(new Integer(empty)) -> storePrimInt(primInt(I - I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB* -> K) =
     k(exp(new Integer(empty)) -> storePrimInt(primInt(I * I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB/ -> K) =
     k(exp(new Integer(empty)) -> storePrimInt(primInt(I quo I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primIntB% -> K) =
     k(exp(new Integer(empty)) -> storePrimInt(primInt(I rem I')) -> K) .

  eq k(val(primInt(I),primFloat(F')) -> primIntB+ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(I) + F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB- -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(I) - F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB* -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(I) * F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primIntB/ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(I) / F')) -> K) .

  eq k(val(primInt(I),primInt(I')) -> primInt> -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I > I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt>= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I >= I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt< -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I < I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt<= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I <= I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I == I')) -> K) .
  eq k(val(primInt(I),primInt(I')) -> primInt!= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(I =/= I')) -> K) .

  eq k(val(primInt(I),primFloat(F')) -> primInt> -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) > F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt>= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) >= F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt< -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) < F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt<= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) <= F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) == F')) -> K) .
  eq k(val(primInt(I),primFloat(F')) -> primInt!= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(float(I) =/= F')) -> K) .

***
*** Primitive arithmetic and relational ops for floats, including
*** coersions from ints on the second arguments.
***
  ops primFloatB+ primFloatB- primFloatB* primFloatB/ : -> ContinuationItem .
  ops primFloat> primFloat>= primFloat< primFloat<= primFloat= primFloat!= : -> ContinuationItem .

  eq k(val(primFloat(F),primFloat(F')) -> primFloatB+ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F + F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB- -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F - F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB* -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F * F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloatB/ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F / F')) -> K) .

  eq k(val(primFloat(F),primInt(I')) -> primFloatB+ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F + float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB- -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F - float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB* -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F * float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloatB/ -> K) =
     k(exp(new Float(empty)) -> storePrimFloat(primFloat(F / float(I'))) -> K) .

  eq k(val(primFloat(F),primFloat(F')) -> primFloat> -> K) =
```

```
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F > F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat>= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F >= F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat< -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F < F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat<= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F <= F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F == F')) -> K) .
  eq k(val(primFloat(F),primFloat(F')) -> primFloat!= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F =/= F')) -> K) .

  eq k(val(primFloat(F),primInt(I')) -> primFloat> -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F > float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat>= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F >= float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat< -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F < float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat<= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F <= float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F == float(I'))) -> K) .
  eq k(val(primFloat(F),primInt(I')) -> primFloat!= -> K) =
     k(exp(new Boolean(empty)) -> storePrimBool(primBool(F =/= float(I'))) -> K) .

***
*** Primitive read/write ops for strings, numeric values, and bools
***
  ops primRead primReadInt primReadFloat primReadBool primWrite primConcat primStrLen : -> ContinuationItem .
  eq t(control(k(val(primString(S)) -> primWrite -> K) CS) TS) output(Sl) = t(control(k(K) CS) TS) output(Sl,S) .
  rl t(control(k(primRead -> K) CS) TS) input(S,Sl) =>
     t(control(k(exp(new String(empty)) -> storePrimString(primString(S)) -> K) CS) TS) input(Sl) .
  rl t(control(k(primReadInt -> K) CS) TS) input(S,Sl) =>
     t(control(k(exp(new Integer(empty)) -> storePrimInt(primInt(trunc(rat(S,10)))) -> K) CS) TS) input(Sl) .
  rl t(control(k(primReadFloat -> K) CS) TS) input(S,Sl) =>
     t(control(k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(S))) -> K) CS) TS) input(Sl) .
  rl t(control(k(primReadBool -> K) CS) TS) input("true",Sl) =>
     t(control(k(exp(new Boolean(empty)) -> storePrimBool(primBool(true)) -> K) CS) TS) input(Sl) .
  rl t(control(k(primReadBool -> K) CS) TS) input("false",Sl) =>
     t(control(k(exp(new Boolean(empty)) -> storePrimBool(primBool(false)) -> K) CS) TS) input(Sl) .
  eq k(val(primString(S),primString(S')) -> primConcat -> K) =
     k(exp(new String(empty)) -> storePrimString(primString(S + S')) -> K) .
  eq k(val(primString(S)) -> primStrLen -> K) = k(exp(new Integer(empty)) -> storePrimInt(primInt(length(S))) -> K) .

***
*** Ops to convert primitive values to strings
***
  op toStringInt : -> ContinuationItem .
  eq k(val(primInt(I)) -> toStringInt -> K) = k(exp(new String(empty)) -> storePrimString(primString(string(I,10))) -> K) .

  op toStringFloat : -> ContinuationItem .
  eq k(val(primFloat(F)) -> toStringFloat -> K) = k(exp(new String(empty)) -> storePrimString(primString(string(F))) -> K) .

  op toStringBool : -> ContinuationItem .
  eq k(val(primBool(true)) -> toStringBool -> K) = k(exp(new String(empty)) -> storePrimString(primString("true")) -> K) .
  eq k(val(primBool(false)) -> toStringBool -> K) = k(exp(new String(empty)) -> storePrimString(primString("false")) -> K) .

***
*** Ops to convert strings to primitive values
***
  op toIntString : -> ContinuationItem .
  eq k(val(primString(S)) -> toIntString -> K) = k(exp(new Integer(empty)) -> storePrimInt(primInt(trunc(rat(S,10)))) -> K) .

  op toFloatString : -> ContinuationItem .
  eq k(val(primString(S)) -> toFloatString -> K) = k(exp(new Float(empty)) -> storePrimFloat(primFloat(float(S))) -> K) .

  op toBoolString : -> ContinuationItem .
  eq k(val(primString("true")) -> toBoolString -> K) = k(exp(new Boolean(empty)) -> storePrimBool(primBool(true)) -> K) .
  eq k(val(primString("false")) -> toBoolString -> K) = k(exp(new Boolean(empty)) -> storePrimBool(primBool(false)) -> K) .
```

```
endm

mod SCALARS-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including SCALARS-SYNTAX .
  including NEW-SEMANTICS .
  including PRIMITIVES-SEMANTICS .

  var I : Int . var F : Float . var B : Bool .
  var C : Char . var S : String . var K : Continuation .

***
*** When we find a scalar, create a new object for it
***
  eq k(exp(i(I)) -> K) = k(exp(new Integer(empty)) -> storePrimInt(primInt(I)) -> K) .
  eq k(exp(f(F)) -> K) = k(exp(new Float(empty)) -> storePrimFloat(primFloat(F)) -> K) .
  eq k(exp(b(B)) -> K) = k(exp(new Boolean(empty)) -> storePrimBool(primBool(B)) -> K) .
  eq k(exp(c(C)) -> K) = k(exp(new Char(empty)) -> storePrimChar(primChar(C)) -> K) .
  eq k(exp(s(S)) -> K) = k(exp(new String(empty)) -> storePrimString(primString(S)) -> K) .

endm

mod CONDITIONAL-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SCALARS-SEMANTICS .
  including CONDITIONAL-SYNTAX .
  including PRIMITIVES-SEMANTICS .

  op skip : -> Stmt .
  op if : Stmt Stmt -> ContinuationItem .

  var E : Exp . vars S S' : Stmt . var V : Value .
  var K : Continuation . var O : Object .

  eq if E then S fi = if E then S else skip fi .
  eq k(stmt(skip) -> K) = k(K) .
  eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
  eq val(o(O)) -> if(S,S') = val(o(O)) -> fetchPrimUnary -> if(S,S') .
  eq val(primBool(true)) -> if(S,S') = stmt(S) .
  eq val(primBool(false)) -> if(S,S') = stmt(S') .
endm

mod LOOP-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SCALARS-SEMANTICS .
  including LOOP-SYNTAX .
  including PRIMITIVES-SEMANTICS .
  including CONDITIONAL-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including SEQUENCE-SYNTAX .

  var X : Name . vars E E' : Exp . var S : Stmt . var K : Continuation .
  var CS : State . var Env : Env . vars V V' : Value .

***
*** The for is actually one of the more confusing features
*** to define. The first equation just splits out the expressions
*** E and E' for evaluation. In the second one, we bind the value of the
*** loop index name (X) and save the prior environment. This is done because,
*** if we have a continue, we want to recover the environment at the start of
*** the loop, including the indexer. The third equation sets up the continue
*** continuation, which increments the indexer, evaluates it, puts in the
*** top value (the E' in for X <- E to E'...), checks to see if the indexer
```

```
*** is still <= the top value, and then runs the body and calls continue if so,
*** but breaks if not. The value V' is used, not E', in case E' has side
*** effects. The main continuation does most of this as well -- checks the
*** indexer to see if it is <= V', etc.
***
  op for : Name Stmt -> ContinuationItem .
  op for : Name Value Stmt -> ContinuationItem .
  eq stmt(for X <- E to E' do S od) = exp(E,E') -> for(X,S) .
  eq control(k(val(V,V') -> for(X,S) -> K) CS) env(Env) =
     control(k(val(V) -> bind(X) -> for(X,V',S) -> wrapenv(Env) -> K) CS) env(Env) .
  eq control(k(for(X,V',S) -> K) CS) env(Env) =
     control(k(pushLStack(K,CS,Env,
       stmt(X <- (X . (n('+))(i(1))) ;) ->
         exp(X) -> val(V') -> toInvoke(n('<=)) ->
   if(S, break ;) ->
     contLStack) -> exp(X) -> val(V') -> toInvoke(n('<=)) -> if(S, break ;) -> contLStack) CS) env(Env) .

***
*** The while loop is much simpler...
***
  eq control(k(stmt(while E do S od) -> K) CS) env(Env) =
     control(k(pushLStack(K,CS,Env,exp(E) -> if(S,break ;) -> contLStack) -> exp(E) -> if(S,break ;) -> contLStack) CS)
     env(Env) .

***
*** A continue will "peek" at the stack, resetting the state and running the continue
*** continuation, but not actually removing the top of the lstack. A break will actually
*** pop this stack, since we are leaving the loop.
***
  eq stmt(continue ;) = contLStack .
  eq stmt(break ;) = popLStack .
endm

mod METHOD-SEMANTICS is
  including STATE-HELPERS .
  including METHOD-SYNTAX .
  including SEND-SYNTAX .
  including DECL-SEMANTICS .
  including STMT-SEMANTICS .

  var M : Method . var Ms : Methods . var X : Name .
  vars Xs Xs' : Names . var Ds : Decls . var S : Stmt .
  var MI : MethodItem . vars IM IM' : IMethod .
  var K : Continuation . var MSet : MethodSet .
  var Vl : ValueList . var E : Exp .

  op mdefs : Methods -> MethodSet .
  eq mdefs(method X ( Xs ) is Ds S end, Ms) =
     mthd(mname(X) mparams(Xs) mdecls(decls(Ds)) mbody(stmt(S) -> val(nil) -> return)) mdefs(Ms) .
  eq mdefs(empty) = empty .

***
*** Semantics for return -- if we hit return, pop the method stack.
***
  op return : -> ContinuationItem .
  eq stmt(return ;) = return .
  eq stmt(return E ;) = exp(E) -> return .
  eq k(val(Vl) -> return -> K) = k(val(Vl) -> popMStack -> K) .
  eq k(return -> K) = k(val(nil) -> popMStack -> K) .

endm

mod CLASS-SEMANTICS is
  including STATE-HELPERS .
  including CLASS-SYNTAX .
  including DECL-SEMANTICS .
  including METHOD-SEMANTICS .

***
```

```
*** Two items -- the first represents syntactic(AST) classes,
*** the second the internal class values that we store
***
  op cdef : Class -> ContinuationItem .
  op cval : IClass -> ContinuationItem .

  vars X X' : Name . var Ds : Decls . var Ms : Methods .
  var MSet : MethodSet . var Xs : Names . var CI : ClassItem .
  var K : Continuation . var IC : IClass . var CS : State .
  var CSet : ClassSet . var TS : State . var Cs : Classes .

  op process : Classes -> ClassSet .
  eq process(class X extends X' is Ds Ms end, Cs) =
     cls(cname(X) pname(X') inheritsSet(emptyNS) flds(decls(Ds)) mthds(mdefs(Ms))) process(Cs) .
  eq process(empty) = empty .
endm

mod SELF-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including SELF-SYNTAX .

  eq exp(self) = getCurrentObject .
endm

mod SUPER-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including SEND-SEMANTICS .
  including SUPER-SYNTAX .

  vars Xm Xc Xc' Xc'' : Name . vars K Km : Continuation .
  vars Xs Xs' : Names .
  var E : Exp . var Es : Exps . var Vl : ValueList .
  var CS : State . var CI : ClassItem . var Cs : ClassSet .
  vars O O' : Object . var Env : Env . var IC : IClass .
  var Ms : MethodSet . var TS : State .

***
*** First, evaluate the call target (which should evaluate to an
*** object) and the parameters (which can be empty).
  op toSuperInvoke : Name -> ContinuationItem .
  op toSuperInvokeCons : -> ContinuationItem .
  eq exp(super. Xm ( Es )) = exp(Es) -> toSuperInvoke(Xm) .
  eq exp(super ( Es )) = exp(Es) -> toSuperInvokeCons .

***
*** Second, now that we have an object back that we can invoke on,
*** save the current context and start searching for the method
*** in the parent class of the current class of the object.
*** Note that, like with normal sends, we clear the lstack so we
*** cannot break or continue out of a method call.
***
  eq t(control(k(val(Vl) -> toSuperInvoke(Xm) -> K) CS) cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc') pname(Xc'') CI) Cs) =
     t(control(k(pushMStack(K,CS,Env,O',Xc') -> clearLStack -> invoke(Xm,Vl)) CS) cclass(Xc'') cobj(O') env(baseenv) TS)
       cset(cls(cname(Xc') pname(Xc'') CI) Cs) .

  eq t(control(k(val(Vl) -> toSuperInvokeCons -> K) CS) cclass(Xc') cobj(O') env(Env) TS)
       cset(cls(cname(Xc') pname(Xc'') CI) Cs) =
     t(control(k(pushMStack(K,CS,Env,O',Xc') -> clearLStack -> invoke(Xc'',Vl)) CS) cclass(Xc'') cobj(O') env(baseenv) TS)
       cset(cls(cname(Xc') pname(Xc'') CI) Cs) .

endm

mod EXCEPTION-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
```

```
  including STMT-SEMANTICS .
  including EXCEPTION-SYNTAX .

  vars S S' : Stmt . vars X Xc : Name . var K : Continuation .
  var CS : State . var Env : Env . var O : Object .
  var E : Exp . var V : Value .

  eq control(k(stmt(try S catch X S' end) -> K) CS) env(Env) cobj(O) cclass(Xc) =
     control(k(pushEStack(K,CS,Env,O,Xc,bind(X) -> stmt(S')) -> stmt(S) -> popEStack) CS) env(Env) cobj(O) cclass(Xc) .

  op toThrow : -> ContinuationItem .
  eq stmt(throw E ;) = exp(E) -> toThrow .
  eq val(V) -> toThrow = val(V) -> popAndRunEStack .

endm

mod TYPECASE-SEMANTICS is
  including STATE-HELPERS .
  including EXP-SEMANTICS .
  including STMT-SEMANTICS .
  including TYPECASE-SYNTAX .
  including CLASS .

  var E : Exp . var Cs : Cases . var C : Case . var EC : ElseCase .
  vars Xc Xc' Xc'' : Name . var O : Object .
  vars CSet CSet' CSet'' : ClassSet .
  vars Xs Xs' Xs'' : NameSet . var S : Stmt .
  var K : Continuation . var CS : State . var CI : ClassItem .
  var TS : State .

  op typecase : Cases -> ContinuationItem .
  op elsecase : ElseCase -> ContinuationItem .
  op noelse : -> ContinuationItem .
  op case : Case -> ContinuationItem .
  op getInherits : Name -> ContinuationItem .
  op buildInherits : Name -> ContinuationItem .
  op inherits : NameSet -> Continuation .
  op discardelse : -> ContinuationItem .

  eq stmt(typecase E of Cs end) = exp(E) -> typecase(Cs) -> noelse .
  eq stmt(typecase E of Cs EC end) = exp(E) -> typecase(Cs) -> elsecase(EC) .

  eq val(o(O myclass(Xc))) -> typecase(Cs) = getInherits(Xc) -> typecase(Cs) .
  eq inherits(Xs) -> typecase(C, Cs) = inherits(Xs) -> case(C) -> typecase(Cs) .
  eq inherits(Xc Xs) -> case(case Xc of S) -> typecase(Cs) = stmt(S) -> discardelse .
  eq inherits(Xc Xs) -> case(case Xc' of S) -> typecase(Cs) = inherits(Xc Xs) -> typecase(Cs) [owise] .
  eq inherits(Xc Xs) -> typecase(empty) = inherits(Xc Xs) .
  eq inherits(Xc Xs) -> elsecase(else S) = stmt(S) .
  eq inherits(Xc Xs) -> noelse -> K = K .
  eq discardelse -> elsecase(EC) -> K = K .
  eq discardelse -> noelse -> K = K .

***
*** If we have already calculated the inherits set, just grab it
*** back out of the class definition.
***
 ceq t(control(k(getInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs) CI) CSet) =
     t(control(k(inherits(Xs) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs) CI) CSet)
  if Xs =/= emptyNS .

***
*** If not, start to calculate it back up towards the root. The advantage here is we
*** can build the entire path at once, so we don't have to revisit this later. Note:
*** we specify Object's inheritsSet in the class definition, so this will not
*** recurse forever or "fall off" the end.
***
 ceq t(control(k(getInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) pname(Xc') inheritsSet(Xs) CI) CSet) =
     t(control(k(getInherits(Xc') -> buildInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) pname(Xc') inheritsSet(Xs) CI) CSet)
  if Xs == emptyNS .
```

```
***
*** When we have calculated the name set back up towards the root, save it
*** and pass it on.
***
  eq t(control(k(inherits(Xs) -> buildInherits(Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs') CI) CSet) =
     t(control(k(inherits(Xs Xc) -> K) CS) TS) cset(cls(cname(Xc) inheritsSet(Xs Xc) CI) CSet) .


endm

mod CONCURRENCY-SEMANTICS is
  including STATE-HELPERS .
  including STMT-SEMANTICS .
  including EXP-SEMANTICS .
  including CONCURRENCY-SYNTAX .
  including SEND-SEMANTICS .
  including SUPER-SEMANTICS .

  var E : Exp . var Es : Exps . var Xm : Name . var Vl : ValueList .
  var K : Continuation . vars CS TS SS : State . var LTS : LockTupleSet .
  var V : Value . var LS : LockSet . var N : Nat .

***
*** When we spawn a new thread, just hand control over to the existing method
*** invocation continuation items. Start with empty stacks, so we cannot
*** "throw" our way out of a thread, for instance.
***
  eq t(control(k(stmt(spawn E ;) -> K) CS) holds(LTS) TS) =
     t(control(k(K) CS) holds(LTS) TS)
     t(control(k(exp(E) -> die) mstack(empty) estack(empty) lstack(empty)) holds(empty) TS) .

***
*** To acquire a lock, we need to check to see if we already hold it. If
*** so, increment the counter. If not, wait until it is available.
***
  op acquire : -> ContinuationItem .
  eq stmt(acquire E ;) = exp(E) -> acquire .
  eq control(k(val(V) -> acquire -> K) CS) holds(LTS [lk(V),N]) =
     control(k(K) CS) holds(LTS [lk(V),s(N)]) .
 crl t(control(k(val(V) -> acquire -> K) CS) holds(LTS) TS) busy(LS) =>
     t(control(k(K) CS) holds(LTS [lk(V),1]) TS) busy(LS lk(V))
  if notin(LS,lk(V)) .

***
*** To release a lock, decrement the counter, unless it is already 1. If
*** it is, remove it from the busy set as well.
***
  op release : -> ContinuationItem .
  eq stmt(release E ;) = exp(E) -> release .
  eq t(control(k(val(V) -> release -> K) CS) holds(LTS [lk(V),1]) TS) busy(LS lk(V)) =
     t(control(k(K) CS) holds(LTS) TS) busy(LS) .
  eq t(control(k(val(V) -> release -> K) CS) holds(LTS [lk(V),s(N)]) TS) busy(LS lk(V)) =
     t(control(k(K) CS) holds(LTS [lk(V),N]) TS) busy(LS lk(V)) [owise] .

***
*** When a thread dies, remove it. Also, release all its
*** locks.
***
  op die : -> ContinuationItem .
  eq t(control(k(val(V) -> die) CS) holds(LTS) TS) busy(LS) = busy(LS - LTS) .

endm

mod PROGRAM-SEMANTICS is
  including STATE-HELPERS .
  including PROGRAM-SYNTAX .
  including CLASS-SEMANTICS .
  including EXP-SEMANTICS .
```

```
  including NEW-SEMANTICS .
  including PRIMITIVES-SEMANTICS .
  including SCALARS-SEMANTICS .
  including SELF-SEMANTICS .
  including SEND-SEMANTICS .
  including NAME-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including BLOCK-SEMANTICS .
  including SEQUENCE-SEMANTICS .
  including CONDITIONAL-SEMANTICS .
  including SUPER-SEMANTICS .
  including EXCEPTION-SEMANTICS .
  including TYPECASE-SEMANTICS .
  including LOOP-SEMANTICS .
  including CONCURRENCY-SEMANTICS .

endm
```

E KOOL: Prelude

```
mod PROGRAM-PRELUDE is
  including PROGRAM-SEMANTICS .

***
*** Predefined methods
***
  ops mprimIntB+ mprimIntB- mprimIntB* mprimIntB/ mprimIntB% : -> IMethod .
  ops mprimInt< mprimInt<= mprimInt> mprimInt>= mprimInt= mprimInt!= : -> IMethod .
  ops mprimInt2Str : -> IMethod .

  eq mprimIntB+ = mthd(mname(n('+)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primIntB+ -> return)) .
  eq mprimIntB- = mthd(mname(n('-)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primIntB- -> return)) .
  eq mprimIntB* = mthd(mname(n('*)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primIntB* -> return)) .
  eq mprimIntB/ = mthd(mname(n('/)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primIntB/ -> return)) .
  eq mprimIntB% = mthd(mname(n('%)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primIntB% -> return)) .
  eq mprimInt2Str = mthd(mname(n('toString)) mparams(empty) mdecls(empty)
                       mbody(exp(self) -> fetchPrimUnary -> toStringInt -> return)) .
  eq mprimInt< = mthd(mname(n('<)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt< -> return)) .
  eq mprimInt<= = mthd(mname(n('<=)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt<= -> return)) .
  eq mprimInt> = mthd(mname(n('>)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt> -> return)) .
  eq mprimInt>= = mthd(mname(n('>=)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt>= -> return)) .
  eq mprimInt= = mthd(mname(n('=)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt= -> return)) .
  eq mprimInt!= = mthd(mname(n('!=)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primInt!= -> return)) .

  ops mprimFloatB+ mprimFloatB- mprimFloatB* mprimFloatB/ : -> IMethod .
  ops mprimFloat2Str : -> IMethod .

  eq mprimFloatB+ = mthd(mname(n('+)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primFloatB+ -> return)) .
  eq mprimFloatB- = mthd(mname(n('-)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primFloatB- -> return)) .
  eq mprimFloatB* = mthd(mname(n('*)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primFloatB* -> return)) .
  eq mprimFloatB/ = mthd(mname(n('/)) mparams(n('n)) mdecls(empty)
                       mbody(exp(self,n('n)) -> fetchPrimBinary -> primFloatB/ -> return)) .
  eq mprimFloat2Str = mthd(mname(n('toString)) mparams(empty) mdecls(empty)
                       mbody(exp(self) -> fetchPrimUnary -> toStringFloat -> return)) .

  op mprimBool2Str : -> IMethod .
  eq mprimBool2Str = mthd(mname(n('toString)) mparams(empty) mdecls(empty)
                     mbody(exp(self) -> fetchPrimUnary -> toStringBool -> return)) .

  ops mObjCons mprimObj2Str : -> IMethod .
  eq mObjCons = mthd(mname(Object) mparams(empty) mdecls(empty) mbody(return)) .
  eq mprimObj2Str = mthd(mname(n('toString)) mparams(empty) mdecls(empty)
                       mbody(exp(s("Object")) -> return)) .

  ops mprimConsWrite mprimConsReadString mprimConsReadInt mprimConsReadFloat mprimConsReadBool : -> IMethod .
  eq mprimConsWrite = mthd(mname(n('<<)) mparams(n('o)) mdecls(empty)
                      mbody(exp(n('o)) -> toInvoke(n('toString)) -> fetchPrimUnary -> primWrite -> exp(self) -> return)) .
  eq mprimConsReadString = mthd(mname(n('readString)) mparams(empty) mdecls(empty)
                               mbody(primRead -> return)) .
  eq mprimConsReadInt = mthd(mname(n('readInteger)) mparams(empty) mdecls(empty)
                            mbody(primReadInt -> return)) .
  eq mprimConsReadFloat = mthd(mname(n('readFloat)) mparams(empty) mdecls(empty)
```

```
                              mbody(primReadFloat -> return)) .
  eq mprimConsReadBool = mthd(mname(n('readBoolean)) mparams(empty) mdecls(empty)
                             mbody(primReadBool -> return)) .
  op mprimStr2Str : -> IMethod .
  eq mprimStr2Str = mthd(mname(n('toString)) mparams(empty) mdecls(empty)
                         mbody(exp(self) -> return)) .

  ops mprimStrConcat mprimStrLen : -> IMethod .
  eq mprimStrConcat = mthd(mname(n('+)) mparams(n('s)) mdecls(empty)
                           mbody(exp(self,n('s)) -> fetchPrimBinary -> primConcat -> return)) .
  eq mprimStrLen = mthd(mname(n('length)) mparams(empty) mdecls(empty)
                        mbody(exp(self) -> fetchPrimUnary -> primStrLen -> return)) .

  ops mprimStrToInt mprimStrToFloat mprimStrToBool : -> IMethod .
  eq mprimStrToInt = mthd(mname(n('toInteger)) mparams(empty) mdecls(empty)
                          mbody(exp(self) -> fetchPrimUnary -> toIntString -> return)) .
  eq mprimStrToFloat = mthd(mname(n('toFloat)) mparams(empty) mdecls(empty)
                            mbody(exp(self) -> fetchPrimUnary -> toFloatString -> return)) .
  eq mprimStrToBool = mthd(mname(n('toBoolean)) mparams(empty) mdecls(empty)
                           mbody(exp(self) -> fetchPrimUnary -> toBoolString -> return)) .

  op mMNFECons : -> IMethod .
  eq mMNFECons = mthd(mname(n('MethodNotFoundException)) mparams(n('x)) mdecls(empty)
                      mbody(exp(n('x)) -> assignTo(n('xname)) -> return)) .
***
*** Predefined classes
***
  ops IntegerClass FloatClass BooleanClass CharClass : -> IClass .
  ops StringClass ObjectClass ConsoleClass : -> IClass .
  ops ExceptionClass NilPointerExceptionClass MethodNotFoundExceptionClass : -> IClass .
  ops InvalidSignatureExceptionClass LockNotHeldExceptionClass : -> IClass .

  eq ObjectClass = cls(cname(Object) flds(empty) mthds(mObjCons mprimObj2Str) inheritsSet(Object)) .
  eq IntegerClass = cls(cname(Integer) pname(Object) flds(pval) inheritsSet(emptyNS)
                        mthds(mprimIntB+ mprimIntB- mprimIntB* mprimIntB/ mprimIntB% mprimInt2Str
      mprimInt< mprimInt<= mprimInt> mprimInt>= mprimInt= mprimInt!=)) .
  eq FloatClass = cls(cname(Float) pname(Object) flds(pval) inheritsSet(emptyNS)
                      mthds(mprimFloatB+ mprimFloatB- mprimFloatB* mprimFloatB/ mprimFloat2Str)) .
  eq BooleanClass = cls(cname(Boolean) pname(Object) flds(pval) inheritsSet(emptyNS) mthds(mprimBool2Str)) .
  eq CharClass = cls(cname(Char) pname(Object) flds(pval) inheritsSet(emptyNS) mthds(empty)) .
  eq StringClass = cls(cname(String) pname(Object) flds(pval) inheritsSet(emptyNS) mthds(mprimStr2Str
                   mprimStrConcat mprimStrLen mprimStrToInt mprimStrToFloat mprimStrToBool)) .
  eq ConsoleClass = cls(cname(Console) pname(Object) flds(empty) inheritsSet(emptyNS) mthds(mprimConsWrite
                    mprimConsReadString mprimConsReadInt mprimConsReadFloat mprimConsReadBool)) .
  eq ExceptionClass = cls(cname(Exception) pname(Object) flds(empty) inheritsSet(emptyNS) mthds(empty)) .
  eq NilPointerExceptionClass = cls(cname(NilPointerException) pname(Exception) flds(empty)
                                inheritsSet(emptyNS) mthds(empty)) .
  eq MethodNotFoundExceptionClass = cls(cname(MethodNotFoundException) pname(Exception) flds(n('xname))
                                    inheritsSet(emptyNS) mthds(mMNFECons)) .
  eq InvalidSignatureExceptionClass = cls(cname(InvalidSignatureException) pname(Exception) flds(empty)
                                      inheritsSet(emptyNS) mthds(empty)) .
  eq LockNotHeldExceptionClass = cls(cname(LockNotHeldException) pname(Exception) flds(empty)
                                 inheritsSet(emptyNS) mthds(empty)) .


***
*** Predefined class set, that can be added to the starting set
***
 op preludeClasses : -> ClassSet .
 eq preludeClasses = IntegerClass FloatClass BooleanClass CharClass StringClass ObjectClass ConsoleClass
                     ExceptionClass NilPointerExceptionClass MethodNotFoundExceptionClass InvalidSignatureExceptionClass
     LockNotHeldExceptionClass .

***
*** Predefined objects -- when we start execution, several have already been
*** created
***
 op console : -> Name .
 op consoleobj : -> Object .
```

```
 eq consoleobj = myclass(Console) oenv([Object,empty][Console,empty]) .

***
*** Starting store, includes information on the predefined objects
***
 op preludeStore : -> Store .
 eq preludeStore = [loc(0),o(consoleobj)] .
 eq baseenv = [n('console),loc(0)] .
endm
```

F KOOL: Main Module

The main module allows the execution of KOOL programs, with termination in a consistent state (i.e. termination is not allowed in the middle of an otherwise normal execution – we should reach an end state first).

```
mod MAIN-SEMANTICS is
  including PROGRAM-SEMANTICS .
  including PROGRAM-PRELUDE .
  including MAIN-SYNTAX .

  var P : Program . vars TS CS S : State .
  var Sl : StringList . var E : Exp . var Cs : Classes .

  op stop : -> ContinuationItem .

  op eval : Program -> StringList .
  op eval* : Program -> StringList .
  op evalI : Program StringList -> StringList .
  op evalI* : Program StringList -> StringList .

  eq eval(Cs E) = getOutput(
              t(control(k(exp(E) -> discard -> stop) mstack(empty) estack(empty) lstack(empty))
              env(baseenv)  cobj(consoleobj) cclass(Console) holds(empty)) input(empty) output(empty)
       mem(preludeStore) nextLoc(1) cset(process(Cs) preludeClasses) busy(empty)
       ) .
  eq eval*(Cs E) = getOutput*(
              t(control(k(exp(E) -> discard -> stop) mstack(empty) estack(empty) lstack(empty))
              env(baseenv)  cobj(consoleobj) cclass(Console) holds(empty)) input(empty) output(empty)
       mem(preludeStore) nextLoc(1) cset(process(Cs) preludeClasses) busy(empty)
       ) .

  eq evalI(Cs E,Sl) = getOutput(
              t(control(k(exp(E) -> discard -> stop) mstack(empty) estack(empty) lstack(empty))
              env(baseenv)  cobj(consoleobj) cclass(Console) holds(empty)) input(Sl) output(empty)
       mem(preludeStore) nextLoc(1) cset(process(Cs) preludeClasses) busy(empty)
       ) .
  eq evalI*(Cs E,Sl) = getOutput*(
              t(control(k(exp(E) -> discard -> stop) mstack(empty) estack(empty) lstack(empty))
              env(baseenv)  cobj(consoleobj) cclass(Console) holds(empty)) input(Sl) output(empty)
       mem(preludeStore) nextLoc(1) cset(process(Cs) preludeClasses) busy(empty)
       ) .

  op getOutput : State -> StringList .
  op getOutput* : State -> StringList .

  eq getOutput(t(control(k(stop) CS) TS) output(Sl) S) = Sl .
  eq getOutput*(t(control(k(stop) CS) TS) input(Sl1:StringList) output(Sl)
     mem(ST:Store) nextLoc(N:Nat) cset(CSet:ClassSet) busy(empty)) = Sl .

endm
```