University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2005-2650

An Executable Semantic Definition of the Beta Language using Rewriting Logic [⋆]

Mark Hills, T. Barış Aktemur, and Grigore Roşu

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

Abstract. In this paper, we present an overview of our method of specifying the semantics of programming languages using rewriting logic. This method, which we refer to as the "continuation-based style", relies on an explicit representation of a program's control context, allowing flexibility in defining complex, control-intensive features of languages while still allowing simple definitions of simple language constructs. To illustrate this technique, we present a definition of a significant subset of the object-oriented language Beta running in the MAUDE rewriting engine. This specification gives us an executable platform for running Beta programs and for experimenting with new language features. We illustrate this by extending the language with super calls. We also touch upon some features of the underlying framework, including the ability to model check Beta programs running on our framework with rewriting-based tools.

keywords: programming language semantics, rewriting logic, Beta

1 Introduction

Rewriting logic provides a flexible framework for specifying the semantics of programming languages[14]. Ideally, we would like our semantic definitions to be simple, modular, and executable. We believe each of these properties is important. Simple definitions are more likely to be used by others, and are more likely to be correct since they are more easily understood. Modular definitions give us the ability to compose language features more effectively, and to more easily extend languages with new features. Executable definitions give us a platform for testing programs written in the language under consideration, lower the turnaround time between coming up with a language feature and having a working interpreter for that feature, and provide an environment for verifying important properties of programs.

The language development methodology presented here is our current attempt to meet these goals, and is an active area of our research. The methodology is based around the explicit representation of control state in the semantics, allowing the straight-forward definition of complex control features such

⋆ Supported by the joint NSF/NASA grant CCR 0234524, NSF CCF-0448501, and NSF CNS-0509321.

as structured jumps (exceptions, for instance) while allowing for simple definitions of truly simple features. We refer to our style of language definition as a "continuation-based style", since we use a continuation-like mechanism to track the control context.

So far, we have used this framework to define a number of languages, both simple languages for pedagogical purposes[20] and more complex languages[4]. This includes a significant subset of the Beta language[12] which we present here. We have organized the rest of the paper as follows. In section 2, we present an overview of rewriting logic and of our language definition framework. Section 3 then provides a quick introduction to the Beta language. Section 4 presents detailed explanations of some of the definitions in our Beta semantics, with a focus on those parts of the definition that we believe are interesting and representative of our technique. Section 5 shows an extension of the Beta language, while Section 6 presents details about some of the analysis capabilities we get essentially for free by using rewriting logic and Maude. Section 7 presents related work, while Section 8 concludes and presents future work.

2 Rewriting Logic Semantics of Programming Languages

Our framework for defining languages can be seen as using a two-layered approach. The lower layer is simply rewriting logic, with the ability to reduce terms to other equivalent terms using equations and make transitions to non-equivalent terms using rules. On top of this we have a number of constructs that provide higher-level concepts used in our language definitions, such as threads, environments, and stores. In the first section we provide a brief introduction to rewriting logic, while in the second we provide an introduction to the concepts and constructs used in our technique.

2.1 Rewriting Logic

Rewriting logic is an extension of equational logic, so it helps to look there first. In equational logic, terms of various *sorts* (types) are formed from uninterpreted function symbols. Functions are of the form $f : w^* \longrightarrow s$, where w^* is 0 or more sorts and s is the result sort. Constants are simply functions with arity 0. The set of sorts and the set of functions form the *signature*, represented as Σ. Since we make use of multiple sorts, this is referred to as a *multi-sorted* signature. There are also variations on this, including unsorted/single-sorted signatures and order-sorted signatures[6].

With a given signature, it is then possible to specify equations which specify when terms are equivalent. This set of equations is referred to as E, giving an equational theory as a pair (Σ, E) of the signature and equations. These equations can be used computationally by treating them as reduction rules which apply from left to right. A term can then be reduced to a normal form by the repeated application of equations.

While this gives us an efficient mechanism for evaluation, one thing to note is that this does not include nondeterminism. Since all terms, from the initial term to its normal form, are equivalent, equational logic does not directly give us a way to represent making a *choice* in the evaluation sequence which could lead to different results. To support this, equational logic can be extended to rewriting logic. In rewriting logic, the theory becomes a triple (Σ, E, R) where the equational theory is augmented with a set of rules R. A rule is interpreted as a transition between states, providing a mechanism for a computation to split along non-equivalent paths. This leads to a natural association of equations with deterministic language features and rules with potentially non-deterministic features. For instance, in a multi-threaded application, if one thread reads from a variable and another writes to the variable, non-equivalent computation can occur based on which operation is executed first, so we use rules to model the memory operation where the two threads compete.

For more details on rewriting logic, see[13].

2.2 Continuation-Based Style

Using rewriting logic, we can provide semantics for programming languages in what we refer to as a "continuation-based style". To do this, we need to provide an abstract infrastructure for the `state` of the system, and we need to provide equations and rules specifying how language constructs modify the state. One component of the state is the continuation, which gives its name to our method of specifying language semantics. The continuation keeps track of the remainder of the computation, and can be saved, restored, and arbitrarily modified, allowing us to model complex control-altering constructs, such as exceptions, loop break and continue, and call/cc in a straight-forward manner.

In general, the abstract system state is represented as a "soup" where various terms "float" together. The most important terms are the environment, which is a lookup table mapping names to memory locations; the store, which maps locations to values; the continuation, mentioned above; and threads, which are used in multi-threaded languages to represent concurrent execution contexts. Without threads, all the other items are considered to be top level items in the state. With threads, each thread contains its own continuation and environment, with a shared store. Based on language requirements, this "soup" can be extended with other terms; examples of this are shown below in Section 4, when we deal with alternation and concurrency.

For a simple example of a definition in this framework, we take the example of plus, shown below.

```
op plus : -> ContinuationItem .
eq k(E + E' ~> K) = k((E,E') ~> plus ~> K) .
eq k((int(I),int(I')) ~> plus ~> K) = k(int(I + I') ~> K) .
```

The first line defines a new operation of sort `ContinuationItem`, the `plus` operation. We use continuation items throughout the definition of a language to keep track of what we are doing explicitly, without needing to rely on the framework to implicitly keep track of the program's control context. The second and third lines define the two equations needed to carry out the plus operation. Note that, when we see something of the form `k(E ~> K)` this means that expression `E` is the current expression, since it is on top of the continuation (`K` represents the part that we aren't concerned about in this operation). In the first equation we state that to sum two expressions, `E` and `E'`, we first need to evaluate both `E` and `E'`. We put the `plus` continuation item on the continuation to mark that we are performing the plus operation, and so need to continue with this later. In the second we state that, if at the top of the continuation we have two integer values `I` and `I'` on top of a `plus` continuation item, we add the two numbers together and put their sum on top of the continuation, at the same time removing the `plus` item since that operation is complete. Here, note that we are not restricted to only looking at the top term of the continuation, but can look multiple terms down.

For a slightly more complex example, we look at the definition of name lookup and assignment shown below, with the first equation used for lookup and the rest for assignment.

```
eq k(X ~> K) env([X,L] Env) mem([L,V] Mem) = k(V ~> K) env([X,L] Env) mem([L,V] Mem) .
op assignTo : NameList -> ContinuationItem .
eq k(E -> X ~> K) = k(E ~> assignTo(X) ~> nothing ~> K) .
eq k((V,Vl) ~> assignTo(X,Xl) ~> K) env([X,L] Env) mem(Mem) =
   k(Vl ~> assignTo(Xl) ~> K) env([X,L] Env) mem(Mem[L <- V]) .
eq k((nil).ValueList ~> assignTo(nil) ~> K) = k(K) .
```

Note here that we make use of the continuation, store, and environment. To retrieve the value of a name `X`, we find the location `L` of the value in the environment. `env([X,L] Env)` means that we match the name/location pair `[X,L]` and then have the rest of the environment `Env`. Similarly, we then match the location `L` and the value `V` in the store, shown as `mem([L,V] Mem)`. Once we get the value, we just put it on top of the continuation.

Assignment works similarly. First, we define a continuation item, `assignTo`, to represent assignment to a list of names. Next, when we want to assign the value of expression `E` to name `X`, we first need to evaluate `E`, and we flag that we want to assign the result to name `X` (a list of length one) by using the `assignTo` continuation item. The next two equations handle the `assignTo` continuation item. In the first, we assume that we have a list of values `V,Vl` on top of the continuation and a list of names `X,Xl` to assign to. The name `X` should be in the environment having some location `L`. With this, we then recursively handle the rest of the value list and name list while updating the memory with the new value, indicated with the memory update operation `Mem[L <- V]` which updates location `L` with value `V`. The final equation is the recursive stop condition, which just indicates that we will continue with whatever remains of the computation in continuation `K`.

One important point to note is that, in a concurrent language, we would need to change the first and third equations to rules. This is required since multiple threads could compete on the same memory location. With rules, we can capture that the ordering of certain events (memory writes) is important to the outcome of a program execution, giving us multiple possible final program states.

Finally, to show how the control context can be easily modified, note the definition of halt: `eq k(halt ~> K) = k(nothing ~> stop)`. Here we want to immediately stop execution of the program, returning a value of `nothing` as the result. We can directly alter the continuation to do this by rewriting a `halt` to the value `nothing` on top of `stop`, which marks the bottom of the continuation. Note we just discard the rest of the continuation in the process. Similarly, we can handle more complex control jumps by popping items off the top of the continuation until we find the one we are looking for, or by saving the continuation and restoring it later. Techniques such as this can be used to define constructs such as loop breaks and exceptions.

3 The Beta Programming Language

The Beta programming language [12] is an object-oriented language descended from Simula 67 [18]. Two key decisions in the design of the Beta language differentiate it from object-oriented languages such as Java[8]:

– Beta intentionally limits the number of abstraction mechanisms available in the language. A general *pattern* mechanism replaces the multiple types of abstractions from other OO languages, such as the typical distinction between classes and methods. Types of patterns are then distinguished by usage, instead of by language syntax. For instance, class patterns are used to generate object instances, while procedure and function patterns are used for method invocation. Control patterns are used to extend the control constructs available in the language (for instance, to add a foreach construct which will process each element of an array). More details on abstraction in Beta are available in[10, 12].
– Beta does not include the `super` keyword available in most other OO languages. It instead includes an `inner` keyword. When a pattern is invoked, control automatically starts at the root of the pattern hierarchy, a designated object named `Object`. A call to `inner` then calls one step down back towards the invoking pattern. See Figure 2 for an example. This views specialization in child patterns as adding, instead of overriding, the functionality in the parent, and more strongly supports the notion of behavioral subtyping[11].

A sample pattern in Beta is illustrated in Figure 1[1]. Note that pattern definitions start with (# and end with #). The first declaration is for a named function pattern `inc` which will add 1 to any integer passed to it. The second declaration is for an integer variable `n`. Then, in the `do` block of the pattern, we initialize the variable `n` to 5, pass it to a new instance of the `inc` pattern (the `&` creates a new instance), and save the result back into `n` (`->` is assignment, and passing

[1] In our current implementation we have slightly different syntax to aid in parsing. For instance, we put one set of parentheses around the list of imperatives that make up the do block. Almost all changes are minor and would be easily automated by a preprocessing step.

parameters is illustrated by assigning the parameters into the instance being invoked). We then assign `n` to `putint`, which has the effect of printing it. The `newline` call prints a newline.

```
(# inc : (# n : @integer;
           enter n
           do n + 1 -> n
           exit n #);
  n : @integer;
  do 5 -> n; n -> &inc -> n;
     n -> putint; newline;
#)
```

Fig. 1. An increment pattern

```
(#
  first : (# do 5 -> putint; newline;
                inner; 15 ->putint;
                newline #);
  second : first (# do 10 -> putint;
                       newline #);
  do &second
#)
```

Fig. 2. An inner call

In Figure 2, a simple use of inner is shown. Note the syntax in Beta for pattern inheritance – pattern `second` inherits from pattern `first`, illustrated as `second : first (# def #)`. When `second` is called, Beta actually invokes the Object pattern, which is the default parent of a pattern when no parent is specified (so `first` inherits from `Object`). The code in `Object` is just an inner call, so the code in `first` is invoked. This prints 5, then calls inner, transferring control down to `second`. `second` prints 10, then returns control to `first`, which prints 15 and then exits.

4 The Rewriting Logic Semantics Definition of Beta

The dynamic semantics of Beta are specified using rewriting logic in the high-performance rewriting system of MAUDE [2]. While this definition is too large to present fully here, we have pulled out several samples which are representative of the whole and which we believe illustrate the power and flexibility of our technique. The executable semantics, and a number of examples, are available on our web page[1].

4.1 For-loop Semantics

In this section we explain how the semantics of the for-loop is implemented. This example is small yet illustrative enough to give a good flavor of the style we are using.

The syntax of a For-loop in BETA is in the form `(for X : E repeat Is for)`, where `X` is the loop variable with an initial value of 1. The body of the loop is the statement list `Is`. The body is repeated for `E` times, the value of `X` being incremented by one at the end of each iteration. We now explain the equations we used to implement this semantics.

We first need to handle the case that we see a For-loop at the top of the continuation. What we need to do is to evaluate `E`, which gives the number of iterations, and put a mark telling that this is a loop. This is very straight-forward to do in the continuation-based style; just put `E` on top of the continuation and it will be evaluated by other equations/rules.

```
  eq k((for X : E repeat Is for) ~> K) env(Env)
   = k(E ~> for(X,Is) ~> Env ~> K) env(Env) .
```

The `for(X,Is)` reminds us that we will use the evaluated value of `E` as the iteration count of the For-loop. We would like to note that the current `Env` is also pushed into the continuation. This is because `X` may shadow any existing name in the environment, and the shadowed name should be recovered after the For-loop is evaluated. This can simply be done by replacing the environment at the end of the loop with the environment we had right before the loop. The equation below takes care of the replacement of the environment.

```
  eq k(Env ~> K) env(Env') = k(K) env(Env) .
```

After we get the value for the number of iterations, we need to bind `X` to 1, and start the iterations:

```
  eq k(int(I) ~> for(X,Is) ~> K)
   = k(int(1) ~> bindTo(X) ~> loop(X,1,I,Is) ~> K) .
```

The second element of the `loop` item keeps the current iteration number, and the third element is the total number of iterations. At each iteration we will compare these two to decide whether the loop should be ended. The following equations cover the two possible cases.

```
 ceq k(loop(X,I,I',Is) ~> K)
   = k(Is ~> discard ~> int(I + 1) ~> assignTo(X) ~> loop(X,I + 1,I',Is) ~> K)
   if I <= I' .
  eq k(loop(X,I,I',Is) ~> K) = k(K) [owise] .
```

In the first iteration the current iteration count is less than the final iteration number, therefore the loop is run once more, by just putting the body on top of the continuation [2]. The body is followed by the continuation items which will increment the value of the loop variable. Finally, the second element of the `loop` item is also updated. The second equation, which covers the case that the loop should be ended, simply discards the continuation item `loop`. We would like to note that these two equations do not have to deal with how the old environment is recovered after the loop is finished, or how the body of the loop is evaluated. These will be handled by other equations.

4.2 Repetition Semantics

One interesting feature provided by Beta is *repetitions*, i.e. arrays. A repetition can be created with an expression in the form (`[E]^T`), where `E` is the size (i.e. the *range*) of the array, and `T` is the type. (`[E]@T`) can be used to create stack-allocated elements, instead of keeping references to heap-allocated objects.

[2] The continuation item `discard` means that we are not interested in what value the body evaluates to.

There are several things that can be done with a repetition: its range can be accessed, it can be extended, it can be re-created with a new size, a subsequence (i.e. a slice) of its elements can be accessed. In order to be able to provide these operations, we define the `rep` operator:

```
op rep : ValueList Declaration Int -> Value .
```

This is the representation of a repetition. Note that it goes to sort `Value`, which means that this closure can be kept in the store. The first element of this closure is a `ValueList`. It is the values of the elements in the repetition. The second element is a `Declaration`, which is the type of the repetition. The last element is the size of the repetition. This representation makes it easy to provide necessary repetition operations. For instance, let's look at how the range access is defined:

```
op range : -> ContinuationItem .
eq k(Re .range ~> K) = k(Re ~> range ~> K) .
eq k(rep(VL, Dec, N) ~> range ~> K) = k(int(N) ~> K) .
```

Suppose `Re` is a repetition. Then, in Beta, `Re.range` gives the size of this array. As seen in the equations above, to get the range, we first evaluate the target. We put the continuation item `range` into the continuation just as a mark that we would like to get the size of the repetition when its value is returned. The second equation handles this case; it grabs the third element of the repetition value obtained, and puts it on top of the continuation for further computation.

When a repetition is to be extended or re-created with a different size, we just use the information we keep in the repetition value. With the knowledge of the type of the repetition, we create a new repetition, and then according to the requested operation we either append this new repetition to the existing one (and update its range), or we replace the old array with the new one. We don't show the related Maude code because of limited space.

4.3 Method Dispatch Semantics

The Beta language contains both static and dynamic methods. Rules for determining which method to invoke are similar to those in other languages. When reference variables are declared, they are declared with an initial type, say `T`. They can then hold an instance of a pattern of type `T` or of a pattern which inherits from type `T`, say `T'`. Only methods declared in `T` or a parent of `T` can be invoked through the reference. If a method, say `m`, is visible in the declaration of `T` and is statically dispatched we want to invoke the closest instance of `m` to `T` declared between `T` and the root of the inheritance hierarchy, `Object`. If `m` is marked as dynamic, we will perform a similar search, but instead we will start at `T'`, the dynamic type of the object instance.

Since pattern creation in Beta is fairly complex, we don't show all the details here – they can be found in modules `LOOKUP-SEMANTICS` and `METHOD-SEMANTICS` in our language definition. Instead, we show the key equations used in dispatch below.

```
eq k(href(T,L,oref(L')) ~> lookup(X) ~> K) =
   k(ldir(L) ~> block(href(T,L,oref(L'))) ~> lookup(X) ~> K) .

eq k(pt(pname(Xc) P) ~> block(href(T,L,oref(L'))) ~> lookup(X) ~> K) =
   k(deref(oref(L')) ~> dispatch(T,L,oref(L'),pt(pname(Xc) P)) ~> lookup(X) ~> K) .

eq k(pt(virtual(true) P) ~> dispatch(T,TL,oref(L'),pt(P')) ~>  lookup(X) ~> K) =
   k(oref(L') ~> lookup(X) ~> K) .

eq k(pt(virtual(false) P) ~> dispatch(T,TL,oref(L'),pt(P')) ~> lookup(X) ~> K) =
   k(pt(virtual(false) P) ~> K) .
```

We use several value and continuation item terms in these equations. The `href` value represents a reference variable, and holds the declared type `T`, the location `L` in the store of the definition of `T`, and the current object reference `oref(L')` held by the variable. The `lookup` item holds a name we are looking up in an object. `ldir(L)` retrieves a value directly from the store from location `L`. `block` is just used internally, to keep adjoining values in the continuation from being collapsed into a value list. The `deref` item retrieves the actual object stored at the location pointed to by `oref(L)`, and `dispatch` keeps track of the information we need for method dispatch: the type `T`, the location of the type definition `TL`, the object reference `oref(L')` we are dispatching to, and the pattern `pt(P')` that defines the type `T`.

With all this in mind, the above equations are now fairly straight-forward. In the first, when we are looking up a name (say `m`) via a reference (say with static type `T`), we first get back the pattern that defines type `T`. In the second, with the pattern for `T` available, we dereference the referenced object and store the information we will need to perform the method dispatch later. An equation not shown uses the dereferenced object to find the first definition of `m` at the level of `T` or above (towards `Object`). The third and fourth equations then trigger the correct dispatch. In the third, if the definition of `m` states that it is virtual, we will again look up `m`, but this time we will use the dynamic type of the referenced object. We discard the dispatch information because we no longer need it to constrain the lookup. In the fourth, where virtual is false, we have already found the pattern that will be instantiated to run the method `m`, so we just return that. In both cases, we expect a pattern to be returned, since the pattern is used to invoke the method.

4.4 Pattern Membership Semantics

In Beta, variables can be queried for the pattern they instantiate. This is called "pattern membership" and can be used to test, for instance, whether an object is an instance of a particular pattern. The expression (`E1## = E2##`) is true, if `E1` and `E2` both are instances of the same pattern. Here, the expressions can be objects, pattern names, or even standalone patterns.

For the implementation of pattern membership, we first define the following equation, where `getPattern` is a continuation item.

```
eq k((E ##) ~> K) = k(E ~> getPattern ~> K) .
```

This equation puts `E` on top of the continuation and marks that we want to take the evaluated value and get back the pattern it is instantiated from. The returned value may be a reference, an object, or a pattern. These cases are handled (in that order) by the following equations.

```
eq k(oref(L) ~> getPattern ~> K) = k(deref(oref(L)) ~> getPattern ~> K) .
eq k(o(Xc, L, OEnv) ~> getPattern ~> K) = k(ldir(L) ~> K) .
eq k(pt(P) ~> getPattern ~> K) = k(pt(P) ~> K) .
```

If the returned value is a reference, we dereference it to get the object being referenced. Note we need to leave the `getPattern` item on the continuation so we know we are still looking for the actual pattern for the expression. If the return value is an object, we look up the pattern used to create the object from the store – the `L` in `o(Xc, L, OEnv)` holds the location of the pattern this object is an instance of, so `ldir(L)` performs a direct lookup from the store at location `L`. Since this gives us back the pattern, we no longer need `getPattern` on the continuation, so we remove it. If the returned value is a pattern, we just remove `getPattern` from the continuation. Since two values can be compared directly for equality, we can then determine if the patterns of two expressions are identical.

We use similar logic to handle pattern variables. In Beta, a pattern variable can hold a pattern or any of its subpatterns, allowing the variable to be used in place of the pattern name to create new instances of the pattern. Since we already have the logic in place to define retrieving a pattern from an expression, we just need to add a small number of equations to handle assigning the retrieved pattern to a variable and then allowing this variable to defined to hold patterns.

4.5 Code as Values Semantics

In Beta it is not only possible to determine the pattern of an expression, it is also possible to treat patterns as values which can be assigned to pattern variables and then instantiated like any other pattern. To do this, we can leverage other parts of the language definition, including definitions for assignment and pattern membership, requiring us to add only 5 equations.

First, we need an equation to handle the declaration of a pattern variable. This equation is shown below.

```
eq k(createDec(## T) ~> K) env(Env [T,L]) =
   k(pvar(T, L, nothing) ~> K) env(Env [T,L]) .
```

This creates a new variable which tracks the original pattern type of the declaration, the location of the pattern type, and the current pattern held by the variable (nothing). Although not currently used, this can be used to ensure that only patterns of type T or children of T can be assigned to this variable.

Next, we can now use a pattern variable in a pattern membership test. This equation handles this case, without requiring us to change the pattern membership module.

```
eq k(pvar(T,L,V) ~> getPattern ~> K) = k(V ~> K) .
```

Third, we need an equation to handle invoking a pattern based on the variable. For this, we can just return the pattern stored in the variable, which will then be used for the invocation.

```
eq k(pvar(T,L,pt(P)) ~> lookupForInvoke ~> K) =
   k(pt(P) ~> lookupForInvoke ~> K) .
```

Finally, we need two equations to handle assigning to a pattern variable. The first handles evaluating both sides of the assignment, while placing a token on the continuation to mark that this is a pattern variable assignment. The second takes the results from the above evaluation, modifies the `pvar`, and assigns it back to the pattern variable.

```
eq k((E -> (X ##)) ~> K) = k((E,X) ~> assignToPVar(X) ~> nothing ~> K) .
eq k((pt(P), pvar(T,L,V)) ~> assignToPVar(X) ~> K) =
   k(pvar(T,L,pt(P)) ~> assignTo(X) ~> K) .
```

4.6 Concurrency Semantics

In Beta, objects are declared as one of two kinds: item kinds or component kinds. Item objects are parts of other objects (including the main program), while component objects are able to run concurrently (or via alternation, discussed in the next section). A variable is declared to hold a component object with special declaration syntax: instead of `v : @ T`, we have `v : @| T`. Concurrent execution of the object held by the variable can then be started with a `fork` call, like `v.fork`. The object will execute until the end of it's do block (when a non-concurrent call would normally return), with the thread then terminating[3].

In our definition of Beta, we can succinctly support this with just 4 additional equations.

```
op tofork : -> ContinuationItem .
eq k((E .fork) ~> K) = k(E ~> tofork ~> K) .
eq t(k(frozenObj(pt(P),Env) ~> tofork ~> K) env(Env') holds(Cs) innerList(Vl) TS) =
   t(k(K) env(Env') holds(Cs) innerList(Vl) TS)
   t(k(pt(P) ~> createForInvoke ~> die) env(Env) holds(empty) innerList(nil) TS) .
eq t(k(Vl ~> die) TS) PLS = PLS .
eq t(k(die) TS) PLS = PLS .
```

The first equation handles the creation of a new thread. When we have an expression we want to fork, we first need to evaluate the expression. We place a `tofork` continuation item on the continuation so we know we want to create a new thread when we have finished evaluating the expression. The third and fourth equations handle the case where a thread terminates, either with or without return values. Since we have no way of returning values directly from a thread, we just discard them along with the rest of the thread structure, leaving just the rest of the state (represented by `PLS`).

The second equation is more complex. At this point, the expression to fork has evaluated to the expected result – a frozen object. This is an object of kind component that was "frozen" during the creation process so it would not run. We now want to unfreeze it in a new thread. The frozen object contains two pieces of information: the pattern `P` that we will use to create it, and the environment `Env`

[3] Concurrent threads in Beta can also be suspended with a suspend call, and restarted with another fork call. We do not currently support suspend for threads, just for alternation. It should be straightforward to add this. Also note that in the Mjolner system, the fork call looks like `a[]->fork`, not `a.fork`. Our syntax is modeled after[12].

under which it should be created. We also have some information in the *thread state*, including the current environment, `env(Env')`, the current set of locks held by the thread, `holds(Cs)`, and the current list of targets for inner calls, `innerList(Vl)`. There is potentially more thread state in `TS` which we don't need to examine. We will rewrite this from one thread structure, `t(...)`, into two. The first will maintain the same information in the thread, except it will discard the frozen object and fork items. The second will take the frozen object's pattern and put that on top of the `createForInvoke` item, which will create a new instance of the object. This will be on top of the `die` item. We will run this in the frozen object's environment, `Env`, with no locks held (`holds(empty)`) and with no inner calls available (since we cannot inner out of the thread, and since creating the object will create a new list of available inner calls anyway). We can then maintain any other parts of the state from the old thread.

4.7 Alternation Semantics

Alternation is similar to concurrency, except instead of creating new threads we alternately run different objects in the same thread. Each object runs until it either finishes its `do` block or issues a `suspend` call. The most interesting part of the alternation semantics deal with suspending an object and restarting it later, so we show part of this process below.

```
eq t(k(saveAndRecoverStack(Vl) ~> K) env(Env) activeAltern(X) TS
     alternState(k(K') env(Env') activeAltern(X') alternState(TS'') TS'))
 = t(k((frozenState(k(K) env(Env) TS)) ~> assignTo X ~> Vl ~> K') env(Env')
     activeAltern(X') alternState(TS'') TS') .
```

This equation is triggered by a `suspend` call; a similar equation handles the restart. When this happens, we want to save the current execution stack and recover the prior stack. We use several pieces of state information to help with this. `activeAltern` contains the name of the variable that holds the alternating object. On the other hand, `alternState` contains the thread state that was current when we started the alternating execution of this object – this is the state we need to get back to on `suspend`. Note that it contains similar information to the current context, including its own version of the `alternState` (so we could have alternating objects inside an alternating object). We then take the current pieces of state we wish to save, including the current continuation (minus the `saveAndRecoverStack` item), current environment, and other parts of the thread state, but not the alternating information (which will be recreated when we run this again), put these into a `frozenState` item, and assign this back to the variable that holds the object we are currently running. We also return exit values (`Vl`), and put back in the old continuation that was in the `alternState`, `K'`. At the same time, we recover the old environment, `activeAltern`, and `alternState`, along with other thread state information. To summarize, we take the current state, save it back into the variable that holds this alternating object, and get back all the state we had before we started the alternation. This ability to directly manipulate the control context allows us to model complex operations such as this in a fairly straight-forward manner.

4.8 Semaphores

Beta provides semaphores as a feature of the language. There are two operations defined on a semaphore: V and P. V is the *increment (i.e. acquire) operation, whereas P is the* decrement (i.e. relase) operation. Semaphores can be used to control how many threads can enter a critical region at the same time. In Beta, the special type `Semaphore` is used to create semaphores. A very simple example is given below.

```
(# s  : @ Semaphore ;
   t1 : @ | (# do (s .P ; *** acquire the lock
                   str("t1 is in critical region...") -> puttext ;
                   s .V)  *** release the lock
       #) ;
   t2 : @ | (# do (s .P ; *** acquire the lock
                   str("t2 is in critical region...") -> puttext ;
                   s .V)  *** release the lock
       #) ;
   do (t1 .fork ; t2 .fork )
#)
```

In order to implement semaphores in our framework, we first define the equations to create and access semaphores:

```
eq k(createDec(@ Semaphore) ~> K) = k(sem(0) ~> K) .
eq k((X .P) ~> K) = k(semaphoreRead(X) ~> semP ~> K) .
eq k((X .V) ~> K) = k(semaphoreRead(X) ~> semV ~> K) .
```

The first equation above returns a semaphore with the initial value 0. V and P operations manipulate this value. The second and third equations start evaluation of V and P. We would like to note that to access a semaphore, which is a value kept in store, regular variable read operation is not used. We define a special tag called `semaphoreRead` for this purpose. The reason is that a regular variable access is done via a rule to allow concurrency; that is, to allow threads to compete for the value, as explained in Section 2.1. However, semaphores are atomicly accessed values by definition. Therefore we use an equation (shown below) for accessing a semaphore. This ensures that semantics of the semaphores is preserved.

```
eq t(k(semaphoreRead(X) ~> K) env([X,L] Env) TS) mem([L,V] Mem)
 = t(k((loc(L),V) ~> K) env([X,L] Env) TS) mem([L,V] Mem) .
```

The `semaphoreRead` equation above returns the semaphore value which is kept in the store along with the location of the value. This location is used to uniquely identify different semaphores.

After a semaphore is read, depending on the operation (P or V), there are different cases that need to be covered. For P operation, the cases are:

- Semaphore value is greater than zero: Value is simply decreased.
- Semaphore value is zero: The thread that's issuing the operations becomes the owner of the semaphore.
- Semaphore value is less than zero, and semaphore is held by current thread: The value is simply decreased.

– Semaphore value is less than zero, and semaphore is not held by current thread: The thread is suspended. Thread starts to wait for the semaphore to be released.

These cases, in the given order, are handled by the equations below:

```
ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds(Is) TS) mem(Mem)
  = t(k(allowContextSwitch ~> K) holds(Is) TS) mem(Mem[loc(N) <- sem(I + (-1))])
  if I > 0 .
 eq t(k((loc(loc(N)), sem(0)) ~> semP ~> K) holds(Is) TS) mem(Mem)
  = t(k(allowContextSwitch ~> K) holds(N # Is) TS) mem(Mem[loc(N) <- sem(-1)]) .
ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds(N # Is) TS) mem(Mem)
  = t(k(allowContextSwitch ~> K) holds(N # Is) TS) mem(Mem[loc(N) <- sem(I + (-1))])
  if I < 0 .
ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds(Is) TS) mem(Mem)
  = t(k(waitingFor(loc(N)) ~> semP ~> K) holds(Is) TS) mem(Mem)
  if I < 0 [owise] .
```

We would like to note the usage of `allowContextSwitch` item. As explained above, we use `semaphoreRead` equation to access semaphores. While ensuring atomic read-write access of a semaphore by a thread, this equation doesn't allow a context switch after a semaphore operation is complete. To make such a switch possible, we put the `allowContextSwitch` item on top of the continuation, which is removed by the following *rule*:

```
rl k(allowContextSwitch ~> K) => k(K) .
```

As for the V operation, the cases that need to be covered are similar to those of P. Because of space constraints we do not give all the four equations we defined, but only show the most interesting case: The semaphore value is -1 and it is being released by its owner thread. This makes the semaphore free, and a thread which was suspended on the semaphore is now notified to acquire the lock and continue its execution.

```
eq t(k((loc(loc(N)), sem(-1)) ~> semV ~> K) holds(N # Is) TS)
   mem(Mem) t(k(waitingFor(loc(N)) ~> K') TS')
 = t(k(allowContextSwitch ~> K) holds(Is) TS)
   mem(Mem[loc(N) <- sem(0)]) t(k((loc(loc(N)), sem(0)) ~> K') TS') .
```

The equation above removes the semaphore from the releasing thread's `holds` list, updates the semaphore's value, and replaces the `waitingFor` tag at the top of the suspended thread's continuation with the location and the value of the semaphore. The second equation listed above among the cases for P will be applied to further continue the awaken thread's computation.

5 Examples

We have a number of examples available for download on our website [1]. Here we just show one example of a class pattern that makes use of a number of features in the language. This example is based on the one from [12, p. 188].

```
'Factorial : @ |
  (# 'T : [100] @ integer ; 'N, 'Top, 'Ret : @ integer ;
     enter 'N
     do ( 1 -> 'Top -> 'T[1] ;
       ins ('Cycle (#
         do ((if ('Top lt 'N)
              // True then ( ('Top + 1,'N) ->
                 ins('ForTo (# enter 'First, 'Last
                               do 'T['Index - 1] * 'Index -> 'T['Index] #)) ;
                 'N -> 'Top) if) ;
            'N + 1 -> 'N ;
            'T['N - 1] -> 'Ret ;
            suspend) #))
        )
     exit 'Ret
  #)
```

This pattern is used to compute the factorial of a given number. Note from the definition that this represents a component object (one that we can use for alternation) and is singular. The factorial is computed as needed, up to $n!$, by keeping track of prior results in a repetition `T` and then calculating up to the number `'N`. We make use of two custom control patterns, one called `'Cycle` and one called `'ForTo`. When we inherit from `'Cycle` the do contents are repeated until we explicitly use a `leave` command to exit the cycle. When we inherit from `'ForTo` the do contents are repeated once for each index value starting with `'First` and ending with `'Last`. When we `suspend`, the exit value is passed back to the initiating process. If invoked like `4 -> altern('Factorial) -> 'F`, we will calculate the factorial of 4, while if invoked like `altern('Factorial) -> 'F` we will calculate the factorial of the next number (in this case 5). Note that we have slightly modified the syntax of Beta to make it easier to parse, converting identifiers to quoted identifiers, adding parens where needed, and wrapping inserted objects with `ins` and alternation calls with `altern`. While we do this manually, this could all be done in a pre-processing step automatically.

6 Extending Beta

One advantage of having an executable definition of a language is that we gain the ability to quickly experiment with new language features. As an example, we decided to add `super` calls to the Beta language, taking our inspiration from a paper by Goldberg, Findler, and Flatt[7] where they added `inner` calls to a Java-like object system in MzScheme. Since Beta has both static and virtual method dispatch, we have started by just adding this to static calls, but we believe it would be fairly straight-forward to also add this feature to dynamic calls. We also have restricted this feature to named patterns only – it doesn't seem as useful to allow this with anonymous patterns, since we cannot use an anonymous pattern as a parent to other patterns.

To add `super` calls, we first added a new type of pattern, a Java pattern, which can be the target of a `super` call. This required adding two syntax operators: one to flag a pattern as a Java pattern, and for the `super` keyword. We then needed to add two operators to the semantics as well, one which acts as a boolean flag in our representation of patterns to specify if the pattern is a Java pattern, and a second for a continuation item used for setting this flag on newly created patterns.

In the language semantics, we changed a total of 9 equations and added 9 more. Of the 9 that were changed, 5 were to initialize patterns to not be Java patterns by default, two were to properly handle building the inner call list used for pattern invocations, one was to trigger method invocations to start at Object if the pattern invocation is not of a Java pattern (standard Beta semantics), and the final one was to flag the `Object` pattern as a standard (not Java) Beta pattern.

Of the 9 that were added, 2 were to enumerate fields in a pattern with the new Java pattern syntax, one was to ensure Java patterns are not added to the list for inner calls, one was to ensure that a pattern invocation of a Java pattern would start at that pattern instead of at Object, two were to handle super calls, and three were to handle setting the Java flag on Java patterns.

Overall, it took roughly 2 hours to add this feature into our definition of Beta. We believe the ability to prototype language changes such as this quickly, without needing to make major changes to the language definition, is a compelling feature of this framework. The files with the extended semantics, with examples with and without `super`, are available on our website[1].

7 Formal Analysis

Along with our ability to execute programs directly in the language semantics, our use of rewriting logic and the Maude tool also makes available some tools for formally analyzing Beta programs. We describe our use of two of these tools below.

7.1 Searching Execution Paths

For nondeterministic programs, it is useful to be able to determine what results a program can generate. We can do this using the Maude `search` capability. With search, we can check to see what values a program can generate, and we also can get information on the path a program took to reach the given result. The search capability is aided by our declaration as rules of parts of the semantics where different threads can compete, such as on memory accesses.

For a first, simple example, note the multithreaded program below.

```
search eval*(
(# p : (# do (str("In p ") -> puttext) #) ;
   q : (# do (str("In q ") -> puttext) #) ;
   a : @ | p ; b : @ | q ;
   do (a .fork ; b .fork ) #) ) =>! ST:[String] .
```

Since we don't know which thread will execute first, we expect one of two results: either we will get a message saying we are in p, and then one saying we are in q, or the reverse. This is exactly what search tells us.

```
Solution 1 (state 117)
states: 119  rewrites: 1302 in 191ms cpu (209ms real) (6782 rewrites/second)
```

```
ST:[StringList] --> "In p In q "

Solution 2 (state 118)
states: 119  rewrites: 1302 in 191ms cpu (210ms real) (6782 rewrites/second)
ST:[StringList] --> "In q In p "
```

We may also want to search to see if our program can return a specific value. For instance, the following program theoretically could return any natural number. We want to see if there is a way for the program to return 100. Since we just need a solution to verify 100 is reachable, we limit search to only one solution by putting 1 in brackets. Also, we use the `such that` notation below to ensure that the variable holds the value we are interested in.

```
search[1] eval((#
  a,c : @ integer ;
  t1 : (# do (1 -> a) #) ;
  t : @ | t1 ;
  do (t .fork ;
      ((L : (if a
               // 0 then (c + 1 -> c ; restart L)
               else (leave L) if) : L)) ;
      c -> putint ) #)) =>! ST::String such that ST::String == "100" .
```

When we search, we discover that there is indeed a way to evaluate `c` to 100.

```
Solution 1 (state 7289)
states: 7298  rewrites: 91582 in 21799ms cpu (22981ms real) (4201
    rewrites/second)
ST::String --> "100"
```

7.2 Model Checking

Having specified the semantics of Beta in Maude not only gives an interpreter for the language, but also a model checker. In this section we explain how we can use Maude's model checker to formally analyze Beta programs.

First, we implement the famous dining philosophers problem. Our implementation uses semaphores and has three philosophers.

```
(# fork1 : @ Semaphore ; fork2 : @ Semaphore ; fork3 : @ Semaphore ;
   phil1 : @ | (# do
     (fork1 .P ; fork2 .P ; *** first take left fork, then right
      str("yum yum...") -> puttext ; *** eat
      fork2 .V ; fork1 .V ) *** leave the forks
   #) ;
   phil2 : @ | (# do
     (fork2 .P ; fork3 .P ; *** first take left fork, then right
      str("yum yum...") -> puttext ; *** eat
      fork3 .V ; fork2 .V ) *** leave the forks
   #) ;
   phil3 : @ | (# do
     (fork3 .P ; fork1 .P ; *** first take left fork, then right
      str("yum yum...") -> puttext ; *** eat
      fork1 .V ; fork3 .V ) *** leave the forks
   #) ;
   do(phil1 .fork ; phil2 .fork ; phil3 .fork )
#)
```

The property we would like to check is whether a program can go into deadlock. We follow the framework presented in [3] to write the model-checker. For this, we define a proposition, called `deadlocked`. A `PLState` satisfies this proposition (i.e. `PLS:PLState |= deadlocked = true`), if there is a thread waiting for a semaphore to become available and there is no active thread. This definition is simply implemented as follows:

```
eq t(k(waitingFor(L) ~> K) TS) PLS |= deadlocked = inDeadlock(filterThreads(PLS)) .
*** filter out non-active threads
eq filterThreads(t(k(stop) TS) PLS) = filterThreads(PLS) .
eq filterThreads(t(k(waitingFor(L) ~> K) TS) PLS) = filterThreads(PLS) .
eq filterThreads(PLS) = PLS [owise] .
*** no deadlock if there is some active thread still in the soup
eq inDeadlock(t(TS) PLS) = false .
eq inDeadlock(PLS) = true [owise] .
```

After defining the satisfaction conditions, we can simply check our program by reducing `modelCheck(dining, [] ~ deadlocked)` [4], where `dining` is the constant that builds the initial state for the dining philosophers problem. With the (`[] ~ deadlocked`) formula we want to check that the program never goes into a deadlock state. On a 1000 MHz Linux machine with 770 MB of memory, MAUDE outputs a counterexample in 68 milliseconds disproving that this program is deadlock-free. A quick study of this counterexample shows that there are three threads, each waiting for a semaphore which is held by another thread: a circular dependency. To remove this dependency we change the implementation slightly. The first philosopher now tries to pick the right fork first, instead of the left fork. Model-checking this program outputs the result `true` in 35 seconds, showing that it is now deadlock-free.

8 Related Work

There are a number of different methods for specifying the semantics of programming languages. These include operational methods such as Plotkin's structural operational semantics[19], denotational methods such as Scott and Strachey's[22], and Mosses's action semantics[16], along with many others. Of the above, our work is most similar to operational semantics, with a similar emphasis on computations represented as a sequence of transitions, in our case between terms in rewriting logic.

There has also been work within semantics on making definitions modular. Two examples of this are Mosses's work on modular structural operational semantics[17] and work by Moggi and others on monad transformers in denotational semantics[15].

Finally, there has been some work on executable definitions of language semantics. Some of these have been based on definitions of language semantics encoded in Prolog, such as in Mosses's work on MSOS[17] and in Slonneger and Kurtz's text[21]. Steele gives an interesting example of building compositional language interpreters in Haskell using monads[9]. Friedman, Wand, and Haynes focus on building interpreters in Scheme for representative subsets of different language paradigms in their text[5].

[4] `[]` is the LTL symbol for "always", and `~` is the symbol for "not", so we are checking to see if we are always not deadlocked.

9 Conclusions and Future Work

In this paper, we presented our framework, referred to as the continuation-based style of language definition, for defining the semantics of programming languages using rewriting logic. We showed how this has been used to define the object-oriented language Beta, and demonstrated an example extension of the language. We also showed examples of tools that we get essentially for free from running our language definition on the Maude rewriting engine, including tools for path search and model checking. We believe that the combination of rewriting logic and our continuation-based style of language definition provides a flexible and fairly natural mechanism for defining even complex features of programming languages. The current specification is made up of 45 modules, with 327 equations and 33 rules.

There are several planned directions for extending this research. First, we would like to add any features of Beta that are not currently in the definition. While the definition is relatively complete, we are missing several features, including exceptions. There are also some limitations on usage of certain language constructs that we would like to eliminate. For instance, we assume `inner` calls apply to the current `do` block, but in Beta they can accept the name of a surrounding block as well.

Beyond Beta, we are also working to improve our framework to make it an even better language *design* environment. This work entails improving our notations and frameworks to make defining language semantics easier and more modular, as well as creating modules of common language features that can be used as the basis for language definitions. We would like to make it fairly easy to quickly "put together" languages that can be used as the basis for prototyping new language features.

References

1. FSL Beta language webpage. http://fsl.cs.uiuc.edu/semantics/beta.
2. Maude website. http://maude.cs.uiuc.edu/.
3. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, volume 71 of *ENTCS*. Elsevier, 2002.
4. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *CAV*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
5. D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2nd edition, 2001.
6. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
7. D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *Proc. of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM Press.

8. J. Gosling, B. Joy, and G. Steele. *The Java Language Definition*. Addison-Wesley, 1996.
9. J. Guy L. Steele. Building interpreters by composing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 472–492, New York, NY, USA, 1994. ACM Press.
10. B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Abstraction mechanisms in the beta programming language. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 285–298, New York, NY, USA, 1983. ACM Press.
11. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
12. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
13. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
14. J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In *Proc. Intl. Joint Conf. on Automated Reasoning IJCAR'04, Cork, Ireland, July 2004*, pages 1–44. Springer LNAI 3097, 2004.
15. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1989.
16. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
17. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, July-December 2004 2004.
18. K. Nygaard and O.-J. Dahl. Simula 67. In R. Wexelblat, editor, *History of Programming Languages*. Addison-Wesley, 1981.
19. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, July-December 2004 2004.
20. G. Roşu. Lecture notes of course on Programming Language Design. Dept. of Computer Science, University of Illinois U.C., 2005. http://www-courses.cs.uiuc.edu/~cs422.
21. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*. Addison-Wesley, 1995.
22. J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* MIT Press, 1977.

A Beta Syntax in Rewriting Logic

```
***********************************
*** SYNTAX OF THE BETA LANGUAGE ***
***********************************

fmod NAME-SYNTAX is
  pr QID .
  sort Name NameList .
  subsort Qid < Name < NameList .
  op _,_ : NameList NameList -> NameList [assoc prec 50] .
  ops a b c d e f g h i j k l m n o p q r s t u v w x y z : -> Name .
  ops Object : -> Name .
```

```
endfm

fmod IMP-SYNTAX is
  pr INT .
  pr NAME-SYNTAX .
  pr STRING .
  sort Imp ImpList .
  subsort Imp < ImpList .
  sort Exp ExpList .
  subsorts Name Int < Exp < ExpList < Imp .
  subsort NameList < ExpList .
  op _;_ : ImpList ImpList -> ImpList [assoc prec 51] .
  op _,_ : ExpList ExpList -> ExpList [assoc prec 50] .
  op str : String -> Exp .
endfm

fmod GENERAL-IMP-SYNTAX is
  pr IMP-SYNTAX .
  op NONE : -> Exp .
  op _->_ : ExpList ExpList -> Exp [assoc prec 40 gather(E e)] .
  op &_ : Exp -> Exp [prec 20] .
  op _[] : Exp -> Exp [prec 25] .
endfm

fmod ARITH-IMP-SYNTAX is
  pr IMP-SYNTAX .
  op _+_ : Exp Exp -> Exp [ditto] .
  op _-_ : Exp Exp -> Exp [ditto] .
  op _*_ : Exp Exp -> Exp [ditto] .
  op _div_ : Exp Exp -> Exp [prec 31] .

  op -_ : Exp -> Exp [ditto] .
endfm

fmod EXPLIST-SYNTAX is
  pr IMP-SYNTAX .
  op _,_ : ExpList ExpList -> ExpList [assoc prec 50] .
  op [_] : ExpList -> Exp .
***  op ((_)) : ExpList -> Exp .
endfm

fmod BOOL-IMP-SYNTAX is
  pr IMP-SYNTAX .
  ops True False : -> Exp .
  op _and_ : Exp Exp -> Exp .
  op _or_ : Exp Exp -> Exp .
  op not_ : Exp -> Exp .
endfm

fmod RELATION-IMP-SYNTAX  is
```

```
  pr IMP-SYNTAX .
  op _eq_ : Exp Exp -> Exp .
  op _neq_ : Exp Exp -> Exp .
  op _lt_ : Exp Exp -> Exp .
  op _leq_ : Exp Exp -> Exp .
  op _gt_ : Exp Exp -> Exp .
  op _geq_ : Exp Exp -> Exp .
endfm

***
*** Repetitions come in two forms. The first is a standard "array"
*** form, such as A[10]. The second is called a slice, and
*** is of the form A[3:9].
***
*** Note that the declaration of a repetition is like:
*** A : [16] @integer; (* static *)
*** or
*** A : [16] ^integer; (* reference *)
***
*** This is handled in the declarations section.
***
fmod REPETITION-IMP-SYNTAX is
  pr IMP-SYNTAX .
  op _[_] : Exp Exp -> Exp [prec 30] .
  op _[_$_] : Exp Exp Exp -> Exp [prec 30] .
  op _.new : Exp -> Exp .
  op _.extend : Exp -> Exp .
  op _.range : Exp -> Exp .
endfm

fmod PATTERN-MEMBERSHIP-IMP-SYNTAX is
  pr IMP-SYNTAX .
  op _## : Exp -> Exp [prec 30] .
endfm

fmod CONTROL-IMP-SYNTAX is
  pr IMP-SYNTAX .
  ---Labels
  sort LabelImp .
  subsort LabelImp < Imp .
***
*** For now, assume we always have surrounding labels,
*** like (l: stuff-here :l), and then we can later
*** treat l: stuff-here as a special case.
*** op _:_ : Name ImpList -> LabelImp [prec 71] .
  op ((_:_:_)) : Name ImpList Name -> LabelImp [prec 70] .
  op leave_ : Name -> Imp .
  op restart_ : Name -> Imp .
endfm
```

```
fmod FOR-SYNTAX is
  pr CONTROL-IMP-SYNTAX .

  op ((for_:_repeat_for)) : Name Exp ImpList -> Exp .
endfm

fmod IF-SYNTAX is
  pr CONTROL-IMP-SYNTAX .

  sorts Case Cases CaseSelection CaseSelections .
  subsort Case < Cases .
  subsort CaseSelection < CaseSelections .

  op __ : Cases Cases -> Cases [assoc prec 39] .
  op //_ : Exp -> CaseSelection [prec 35] .
  op __ : CaseSelections CaseSelections -> CaseSelections [assoc prec 37] .
  op _then_ : CaseSelections ImpList -> Case [prec 38] .
  op ((if__else_if)) : Exp Cases ImpList -> Exp [prec 40] .
  op ((if__if)) : Exp Cases -> Exp [prec 40] .

endfm

fmod ALL-IMP-SYNTAX is
  pr GENERAL-IMP-SYNTAX .
  pr ARITH-IMP-SYNTAX .
  pr BOOL-IMP-SYNTAX .
  pr EXPLIST-SYNTAX .
  pr RELATION-IMP-SYNTAX .
  pr REPETITION-IMP-SYNTAX .
  pr PATTERN-MEMBERSHIP-IMP-SYNTAX .
  pr CONTROL-IMP-SYNTAX .
  pr FOR-SYNTAX .
  pr IF-SYNTAX .
endfm

fmod ATTRIBUTE-SYNTAX is
  pr NAME-SYNTAX .
  sort Attribute Declaration .
  op _:_ : NameList Declaration -> Attribute [prec 60] .
endfm

fmod ATTRIBUTE-PART-SYNTAX is
  pr ATTRIBUTE-SYNTAX .
  sort AttributePart .
  subsort Attribute < AttributePart .
  op _;_ : AttributePart AttributePart -> AttributePart [assoc comm prec 65] .
endfm

fmod ACTION-PART-SYNTAX is
  pr NAME-SYNTAX .
  pr ALL-IMP-SYNTAX .
  sort ActionPart .
  op enter_do_exit_ : NameList ImpList NameList -> ActionPart .
  op enter_do_ : NameList ImpList -> ActionPart .
  op do_exit_ : ImpList NameList -> ActionPart .
  op do_ : ImpList -> ActionPart [prec 52] .
endfm

fmod OBJECT-DESCRIPTOR-SYNTAX is
  pr ATTRIBUTE-PART-SYNTAX .
  pr ACTION-PART-SYNTAX .
  sorts ObjDesc ObjMain .
  subsort ObjMain < ObjDesc .

  op ((#_;_#)) : AttributePart ActionPart -> ObjMain  .
  op ((#_;`#)) : AttributePart -> ObjMain .
  op ((#_#)) : ActionPart -> ObjMain .
  op __ : Name ObjMain -> ObjDesc .
endfm

fmod BETA-TYPES is
  pr NAME-SYNTAX .
  pr OBJECT-DESCRIPTOR-SYNTAX .

  sorts BuiltinType Type .
  subsorts Name BuiltinType < Type .

  ops integer char text : -> BuiltinType .
endfm

fmod DECLARATION-SYNTAX is
  pr OBJECT-DESCRIPTOR-SYNTAX .
  pr BETA-TYPES .

  sorts ObjDecl Ref StaticRef DynamicRef PatternDecl .
  subsorts StaticRef DynamicRef < Ref .
  subsorts Ref ObjDecl < Declaration .
  subsort PatternDecl < Attribute .
  op @_ : ObjDesc -> ObjDecl [prec 0] .
  op @_ : Type -> StaticRef [prec 0] .
  op ^_ : Type -> DynamicRef [prec 0] .

  ---below are for repetition (i.e. arrays)
  op [_]@_ : Exp ObjDesc -> ObjDecl .
  op [_]@_ : Exp Type -> StaticRef .
  op [_]^_ : Exp Type -> DynamicRef .

  --- pattern variables
  op ##_ : Type -> Declaration .
  --- Pattern declaration
  op _:_ : Name ObjDesc -> PatternDecl [prec 60] .

  --- Virtual Pattern declaration
  op _:<_ : Name Name -> PatternDecl [prec 60] .
  op _:<_ : Name ObjDesc -> PatternDecl [prec 60] .
  op _::<_ : Name Name -> PatternDecl [prec 60] .
  op _::<_ : Name ObjDesc -> PatternDecl [prec 60] .

  --- Object insertion
  op ins : Exp -> Exp .
  op ins : ObjDesc -> Exp .
endfm

fmod IO-SYNTAX is
  pr IMP-SYNTAX .

  ops puttext putint putboolean newline : -> Exp .
endfm

fmod CALL-SYNTAX is
  pr DECLARATION-SYNTAX .

  op _._ : Exp Name -> Exp [prec 18 gather(E e)] .
  op this : Name -> Exp .
  op inner : -> Exp .
  op INNER : -> Exp .
  eq INNER = inner .
  op inner_ : Name -> Exp .
  op INNER_ : Name -> Exp .
  eq INNER(N:Name) = inner(N:Name) .
  op origin : -> Name .
endfm

fmod CONCURRENT-OBJECT-SYNTAX is
  pr CALL-SYNTAX .

  sorts ConObjDecl ConStaticRef ConDynamicRef .
  subsort ConObjDecl < Declaration .
  subsorts ConStaticRef ConDynamicRef < Ref .

  op @`|_ : ObjDesc -> ConObjDecl [prec 0] .
  op @`|_ : Type -> ConStaticRef [prec 0] .
  op ^`|_ : Type -> ConDynamicRef [prec 0] .

  op _.fork : Exp -> Exp .
  op suspend : -> Exp .
  op altern : Exp -> Exp .
  op Semaphore : -> BuiltinType .
  op _.P : Exp -> Exp .
  op _.V : Exp -> Exp .
endfm

fmod BETA-SYNTAX is
  pr DECLARATION-SYNTAX .
  pr IO-SYNTAX .
  pr CALL-SYNTAX .
  pr CONCURRENT-OBJECT-SYNTAX .
endfm

parse(
(#

  'Account :
    (# 'balance : @ integer ;

       'Deposit :
         (# 'amount, 'temp : @ integer ;
    enter 'amount
    do 'amount + 'balance -> 'balance
    exit 'balance
          #) ;

       'Withdraw :
         (# 'amount : @ integer ;
    enter 'amount
    do 'balance - 'amount -> 'balance
    exit 'balance
          #) ;
     #) ;

  'account1, 'account2, 'account3 : @ 'Account ;

  'K1, 'K2, 'K3 : @ integer ;

  do(
    100 -> & 'account1 . 'Deposit ;
    200 -> & 'account2 . 'Deposit ;
    300 -> & 'account3 . 'Deposit ;

    150 -> & 'account1 . 'Deposit -> 'K1 ;
    90 -> & 'account3 . 'Withdraw -> 'K3 ;
    90 -> & 'account2 . 'Deposit -> 'K2 ;
    90 -> & 'account3 . 'Withdraw -> 'K3  )

#)
) .

set print with parentheses on .
```

```
parse(& 'Account [] -> 'A1 []) .

set print with parentheses off .

parse(
  'Account :
    (# 'balance : @ integer ;

       'Deposit :
         (# 'amount, 'temp : @ integer ;
    enter 'amount
    do 'amount + 'balance -> 'balance
    exit 'balance
          #) ;

       'Withdraw :
         (# 'amount : @ integer ;
    enter 'amount
    do 'balance - 'amount -> 'balance
    exit 'balance
          #) ;
     #)
) .

parse(  'account1, 'account2, 'account3 : @ 'Account ) .

parse( @ integer ) .

parse( ^ integer ) .

parse( ^ text ) .
```

B Beta Semantics in Rewriting Logic

```
in beta-syntax .

fmod INT-LIST is
  protecting INT .

  sort IntList .
  subsort Int < IntList .

  op nil : -> IntList .
  op hd : IntList -> Int .
  op tl : IntList -> IntList .
  op nil? : IntList -> Bool .
  op _::_ : IntList IntList -> IntList [assoc prec 80 id: nil] .
  op _++_ : IntList IntList -> IntList [assoc prec 82].
  vars IL IL' : IntList . var I I' : Int .

  eq hd(I :: IL) = I .
  eq tl(I :: IL) = IL .
  eq nil?(nil) = true .
  eq nil?(IL) = false [owise] .
  eq IL ++ IL' = IL :: IL' .

endfm

fmod STRING-LIST is
  protecting STRING .

  sort StringList .
  subsort String < StringList .

  op nilS : -> StringList .
  op hd : StringList -> String .
  op tl : StringList -> StringList .
  op nil? : StringList -> Bool .
  op _::_ : StringList StringList -> StringList [assoc prec 80 id: nilS] .
  op _++_ : StringList StringList -> StringList [assoc prec 82] .
  op concat : StringList -> String .

  vars SL SL' : StringList . var S S' : String .

  eq hd(S :: SL) = S .
  eq tl(S :: SL) = SL .
  eq nil?(nilS) = true .
  eq nil?(SL) = false [owise] .
  eq SL ++ SL' = SL :: SL' .

  eq concat(nilS) = "" .
  eq concat(S) = S .
  eq concat(S :: SL) = S + concat(SL) .
endfm

fmod LOCATION is
  including INT .
  sort Location .
  op loc : Nat -> Location .
  op noLoc : -> Location .
endfm

fmod LOCATION-LIST is
  including LOCATION .
  sort LocationList .
  subsort Location < LocationList .
  op nil : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: nil] .
  op locs : Nat Nat -> LocationList .
  vars N # : Nat .
  eq locs(N, 0) = nil .
  eq locs(N, #) = loc(N), locs(N + 1, # - 1) .
endfm

fmod ENVIRONMENT is
  including NAME-SYNTAX .
  including LOCATION-LIST .
  sort Env .
  op empty : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: empty] .
  op _[_] : Env Name -> Location .
  op _[_<-_] : Env NameList LocationList -> Env .
  op _%%_ : Name Env -> Bool .
  var X : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq ([X,L] Env)[X] = L .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
  eq Env[X <- L] = (Env [X,L]) [owise] .
  eq X %% ( [X,L] Env ) = true .
  eq X %% Env = false [owise] .
endfm

fmod OBJ-ENVIRONMENT is
  including ENVIRONMENT .
  sort ObjEnv .
  op empty : -> ObjEnv .
  op [_,_] : Name Env -> ObjEnv .
  op __ : ObjEnv ObjEnv -> ObjEnv [assoc comm id: empty] .
  op _%_%%_ : Name Name ObjEnv -> Bool .
  vars X  Xc : Name . var L : Location . var Env : Env .
  var OEnv : ObjEnv .
  eq X % Xc %% ([Xc, [X,L] Env] OEnv) = true .
  eq X % Xc %% OEnv = false [owise] .
endfm

fmod VALUE is
  sorts Value PreValue .
  subsort Value < PreValue .
  op nothing : -> Value .
endfm

fmod VALUE-LIST is
  including VALUE .
  including INT .
  sorts ValueList PreValueList .
  subsorts Value < ValueList < PreValueList .
  subsort PreValue < PreValueList .
  op nil : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: nil] .
  op _,_ : PreValueList PreValueList -> PreValueList [assoc id: nil] .
endfm

fmod SLICE-SEMANTICS is
  including VALUE-LIST .

*** Auxiliary operations:
  vars I I' N : Int . vars V V' : PreValue . vars VL VL' : PreValueList .
  op getSlice : PreValueList Int Int -> PreValueList .
  op getSlice : Int PreValueList Int Int -> PreValueList .
  eq getSlice(VL, I, I') = getSlice(1, VL, I, I') .
 ceq getSlice(N, (V, VL), I, I') = getSlice((N + 1), VL, I, I') if N < I .
 ceq getSlice(N, VL, I, I') = nil if N > I' .
  eq getSlice(N, (V, VL), I, I') = V, getSlice((N + 1), VL, I, I') [owise] .

  op setSlice : PreValueList Int Int PreValueList -> PreValueList .
  op setSlice : Int PreValueList Int Int PreValueList -> PreValueList .
  eq setSlice(VL, I, I', VL') = setSlice(1, VL, I, I', VL') .
 ceq setSlice(N, (V, VL), I, I', VL') = V, setSlice((N + 1), VL, I, I', VL')
     if N < I .
 ceq setSlice(N, VL, I, I', VL') = VL if N > I' .
  eq setSlice(N, (V, VL), I, I', (V', VL')) = V',
       setSlice((N + 1), VL, I, I', VL') [owise] .
endfm

fmod OBJECT is
  including OBJ-ENVIRONMENT .
  including OBJECT-DESCRIPTOR-SYNTAX .
  including VALUE .
  including BETA-TYPES .
  sort Object  .
  subsort Object < Value .

***
*** We should move the object notation to a "soup" notation like
*** that of the pattern, although we need more "stuff" in the
*** pattern right now. Currently, an object is represented as
*** a triple which includes the name of the defining pattern,
*** the location in the store of the pattern definition, and
*** the local environment for the object. We store the pattern
*** location as well as the name to differentiate between
*** patterns with the same name that are defined in different
*** scopes, and to facilitate lookup of patterns that are
*** out of scope (& a.b will invoke b, but b may be defined
*** within a, so it would not be in scope at the invocation site).
***
```

```
  op o : Name Location ObjEnv -> Object .

***
*** To be consistent, all objects are given names. If they
*** are defined without names, we give them a fresh anonymous
*** name.
***
  op anonObj : Nat -> Name .

***
*** A reference to an object, which can also be stored.
*** Retrieval would then involve dereferencing and looking
*** up the object at the resulting location.
***
  op oref : Location -> Value .

***
*** A reference holder -- used for variables defined
*** like a : ^ b. This way we can keep track of the defined
*** type of a, while also tracking the current referenced
*** location. The triple includes:
*** type name x location of type definition x oref value
***
  op href : Type Location Value -> Value .

endfm

fmod STORE is
  including LOCATION-LIST .
  including VALUE-LIST .
  sort Store .
  op empty : -> Store .
  op [_,_] : Location PreValue -> Store .
  op __ : Store Store -> Store [assoc comm id: empty] .
  op _[_] : Store Location -> PreValue .
  op _[_<-_] : Store LocationList PreValueList -> Store .
  var L : Location .  var Mem : Store .  vars Pv Pv' : PreValue .
  var Ll : LocationList .  var Pvl : PreValueList .
  eq ([L,Pv] Mem)[L] = Pv .
  eq Mem[nil <- nil] = Mem .
  eq ([L,Pv] Mem)[L,Ll <- Pv',Pvl] = ([L,Pv'] Mem)[Ll <- Pvl] .
  eq Mem[L,Ll <- Pv',Pvl] = (Mem [L,Pv'])[Ll <- Pvl] [owise] .
endfm

fmod CONTINUATION is
  sorts Continuation ContinuationItem .
  op stop : -> Continuation .
  op _~>_ : ContinuationItem Continuation -> Continuation .
endfm
```

```
fmod BETA-STATE is
  sorts PLStateAttribute PLState PLThreadStateAttribute PLThreadState .
  subsort PLStateAttribute < PLState .
  subsort PLThreadStateAttribute < PLThreadState .
  including ENVIRONMENT .
  including STORE .
  including CONTINUATION .
  including INT-LIST .
  including STRING-LIST .
  op empty : -> PLState .
  op __ : PLState PLState -> PLState [assoc comm id: empty] .
  op empty : -> PLThreadState .
  op __ : PLThreadState PLThreadState -> PLThreadState [assoc comm id: empty] .
  op k : Continuation -> PLThreadStateAttribute .
  op nextLoc : Nat -> PLStateAttribute .
  op mem : Store -> PLStateAttribute .
  op env : Env -> PLThreadStateAttribute .
  op input : StringList -> PLStateAttribute .
  op output : StringList -> PLStateAttribute .
  op t : PLThreadState -> PLStateAttribute .
endfm

fmod OO-BETA-STATE is
  including BETA-STATE .
  including OBJECT .
  op obj : Object -> PLThreadStateAttribute .
  op class : Value -> PLThreadStateAttribute .
  op nextAnon : Nat -> PLStateAttribute .
  op innerList : ValueList -> PLThreadStateAttribute .
endfm

mod BETA-HELPING-OPERATIONS is
  including NAME-SYNTAX .
  including IMP-SYNTAX .
  including OO-BETA-STATE .
  including BETA-TYPES .

  var X : Name .  vars E E' : Exp .  var El : ExpList .
  var K : Continuation .  vars V V' : Value .  vars Vl Vl' : ValueList .
  var Xl : NameList .  var TS : PLThreadState .
  vars Env Env' : Env .  var Mem : Store .  var N : Nat .
  vars T T' : Type . vars L L' L'' PL : Location .

  op _~>_ : ExpList Continuation -> Continuation .
  op _~>_ : ImpList Continuation -> Continuation .
  op _~>_ : ValueList Continuation -> Continuation .
  eq k((E,El) ~> K) = k(E ~> El ~> K) .
  eq k(Vl ~> El ~> K) = k(El ~> Vl ~> K) .
  eq k(Vl' ~> Vl ~> K) = k(Vl,Vl' ~> K) .
```

```
op _~>_ : Env Continuation -> Continuation .
eq k(Vl ~> Env ~> K) env(Env') = k(Vl ~> K) env(Env) .
eq k(Env ~> K) env(Env') = k(K) env(Env) .

op discard : -> ContinuationItem .
eq k(Vl ~> discard ~> K) = k(K) .

op bindTo_ : NameList -> ContinuationItem .
eq t(k(V,Vl ~> bindTo(X,Xl) ~> K) env(Env) TS)
   mem(Mem) nextLoc(N)
 = t(k(Vl ~> bindTo(Xl) ~> K) env(Env[X <- loc(N)]) TS)
   mem(Mem [loc(N),V]) nextLoc(N + 1) .
eq t(k(V ~> bindTo(X) ~> K) env(Env) TS)
   mem(Mem) nextLoc(N)
 = t(k(K) env(Env[X <- loc(N)]) TS)
   mem(Mem [loc(N),V]) nextLoc(N + 1) .

op emptyNL : -> NameList .

op bindToNothing_ : NameList -> ContinuationItem .
eq t(k(bindToNothing(X,Xl) ~> K) env(Env) TS) nextLoc(N)
 = t(k(bindToNothing(Xl) ~> K) env(Env[X <- loc(N)]) TS)  nextLoc(N + 1) .
eq t(k(bindToNothing(X) ~> K) env(Env) TS) nextLoc(N)
 = t(k(K) env(Env[X <- loc(N)]) TS) nextLoc(N + 1) .
eq t(k(bindToNothing(emptyNL) ~> K) env(Env) TS) nextLoc(N)
 = t(k(K) env(Env) TS) nextLoc(N) .

***
*** The same as bind to nothing, but we return the reference location
*** as well. This is useful when we need to pre-allocate storage and
*** know what location was chosen.
***
op bindToNothingWLoc_ : Name -> ContinuationItem .
op lref : Location -> Value .
eq t(k(bindToNothingWLoc(X) ~> K) env(Env) TS) nextLoc(N)
 = t(k(lref(loc(N)) ~> K) env(Env[X <- loc(N)]) TS) nextLoc(N + 1) .

op assignTo_ : NameList -> ContinuationItem .
rl t(k((V,Vl) ~> assignTo(X,Xl) ~> K) env([X,L] Env) TS) mem(Mem)
=> t(k(Vl ~> assignTo(Xl) ~> K) env([X,L] Env) TS) mem(Mem[L <- V]) .
rl t(k((V) ~> assignTo(X) ~> K) env([X,L] Env) TS) mem(Mem)
=> t(k(K) env([X,L] Env) TS) mem(Mem[L <- V]) .

***
*** We assume here that we are assigning into a location that can hold
*** a reference. If not, this should be caught statically (a static checker
*** is also being developed). Note we just revert to assignTo, but this
*** gives us a hook to do any additional work if we need to. Also note that
*** we throw the type away from the object ref -- we instead keep the type
*** that X will treat this as (which was the defined type for X), so we can
```

```
  *** correctly handle static/dynamic lookups.
  ***
  op assignToRef : Name -> ContinuationItem .
  eq k((oref(L),href(T,PL,V)) ~> assignToRef(X) ~> K) =
     k(href(T,PL,oref(L)) ~> assignTo(X) ~> K) .

  ***
  *** Direct lookup operation -- given a location, get back the value
  *** at this location. This allows for quicker dereferencing.
  ***
  op ldir : Location -> ContinuationItem .
  rl t(k(ldir(L) ~> K) TS) mem(Mem [L,V]) =>
     t(k(V ~> K) TS) mem(Mem [L,V]) .

  ***
  *** Direct assignment operation -- given a value and a location,
  *** put that value in the location.
  ***
  op lassign : Location Value -> ContinuationItem .
  rl t(k(lassign(L,V) ~> K) TS) mem(Mem [L,V']) =>
     t(k(K) TS) mem(Mem [L,V]) .

endm

fmod PATTERN is
  including DECLARATION-SYNTAX .
  including VALUE .
  including ENVIRONMENT .

  sort Pattern . sort PatternItem .
  subsort PatternItem < Pattern .

  op empty : -> Pattern .
  op __ : Pattern Pattern -> Pattern [comm assoc id: empty] .
  op pt : Pattern -> Value .

  op pname : Name -> PatternItem .
  op entryList : NameList -> PatternItem .
  op exitList : NameList -> PatternItem .
  op doCode : ImpList -> PatternItem .
  op parent : Name -> PatternItem .
  op attributes : AttributePart -> PatternItem .
  op pEnv : Env -> PatternItem .
  op pLoc : Location -> PatternItem .
  op parentLoc : Location -> PatternItem .
  op virtual : Bool -> PatternItem .
endfm

mod CANONICALIZE-PATTERN is
  including PATTERN .
  including OO-BETA-STATE .
  including BETA-HELPING-OPERATIONS .

  op canonObj : Name ObjDesc Env Location -> ContinuationItem .
  op canon : ObjMain -> ContinuationItem .
  op canonAtOnly : -> ContinuationItem .
  op canonAP : ActionPart -> ContinuationItem .
  op addNameEnv : Name Name Env Location -> ContinuationItem .

  op attrList : AttributePart -> ContinuationItem .
  op canonAttr : AttributePart -> ContinuationItem .
  op canonAttrHolder : AttributePart -> ContinuationItem .
  op emptyImp : -> ImpList .
  op emptyAtP : -> AttributePart .
  op fields : AttributePart -> NameList .

*** A designated name for the top of the object hierarchy -- we can use
*** this as our "stop" case when recursing back up the inheritance
*** graph.
  op HierTop : -> Name .

  vars AP AP' : ActionPart . vars AtP AtP' : AttributePart .
  vars Ns Ns' : NameList . var Is : ImpList . var OM : ObjMain .
  vars X XcO : Name . var V : Value . var Mem : Store . var N : Nat .
  var Env : Env . vars L L' XL : Location .
  vars Xc Xc' : Name .  var K : Continuation .
  vars Vl Vl' Vl'' : ValueList .  vars O O' : Object .  var OEnv : ObjEnv .

***
*** First step in canonicalization -- put the attributes into a canonical
*** form.
***
  eq k(canonObj (XcO, Xc OM, Env, L) ~> K) =
     k(canon(OM) ~> addNameEnv(XcO, Xc, Env, L) ~> K) .
  eq k(canonObj (XcO, OM, Env, L) ~> K) =
     k(canon(OM) ~> addNameEnv (XcO, Object, Env, L) ~> K) .

  eq k(canon( (# AtP ; #) ) ~> K) =
     k(canonAttr(AtP) ~> canonAttrHolder(emptyAtP) ~> canonAtOnly ~> K) .
  eq k(canon( (# AtP ; AP #) ) ~> K) =
     k(canonAttr(AtP) ~> canonAttrHolder(emptyAtP) ~> canonAP(AP) ~> K) .
  eq k(canon( (# AP #) ) ~> K) =
     k(attrList(emptyAtP) ~> canonAP(AP) ~> K) .

***
*** Once attributes are in canonical form, do this with the rest of the
*** definition as well so we have a consistent one below. Note that we
*** grab the location here for the parent pattern -- we can use this
*** later during layered object creation.
***
```

```
  eq k(attrList(AtP) ~> canonAtOnly ~> addNameEnv(XcO, X, Env [X,XL], L) ~> K) =
     k(pt(pname(XcO) parent(X) attributes(AtP) entryList(emptyNL) parentLoc(XL)
       doCode(emptyImp) exitList(emptyNL) pEnv(Env [X,XL]) pLoc(L)
       virtual(false)) ~> K) .
  eq k(attrList(AtP) ~> canonAP(enter Ns do Is exit Ns') ~>
       addNameEnv(XcO, X, Env [X,XL], L) ~> K) =
     k(pt(pname(XcO) parent(X) attributes(AtP) entryList(Ns) parentLoc(XL)
       doCode(Is) exitList(Ns') pEnv(Env [X,XL]) pLoc(L) virtual(false)) ~> K) .
  eq k(attrList(AtP) ~> canonAP(enter Ns do Is) ~>
       addNameEnv(XcO, X, Env [X,XL] , L) ~> K) =
     k(pt(pname(XcO) parent(X) attributes(AtP) entryList(Ns) parentLoc(XL)
       doCode(Is) exitList(emptyNL) pEnv(Env [X,XL]) pLoc(L) virtual(false)) ~> K) .
  eq k(attrList(AtP) ~> canonAP(do Is exit Ns) ~>
       addNameEnv(XcO, X, Env [X,XL] , L) ~> K) =
     k(pt(pname(XcO) parent(X) attributes(AtP) entryList(emptyNL) parentLoc(XL)
       doCode(Is) exitList(Ns) pEnv(Env [X,XL]) pLoc(L) virtual(false)) ~> K) .
  eq k(attrList(AtP) ~> canonAP(do Is) ~> addNameEnv(XcO, X, Env [X,XL] , L) ~> K) =
     k(pt(pname(XcO) parent(X) attributes(AtP) entryList(emptyNL) parentLoc(XL)
       doCode(Is) exitList(emptyNL) pEnv(Env [X,XL]) pLoc(L) virtual(false)) ~> K) .

***
*** Canonicalize the attributes -- essentially, eliminate lists, so we
*** have one name per type in the decl
***
  var D : Declaration . var P : PatternDecl . var OD : ObjDesc .
  eq k(canonAttr( P ; AtP) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttr(AtP) ~> canonAttrHolder(AtP' ; P) ~> K) .
  eq k(canonAttr(P) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttrHolder(AtP' ; P) ~> K) .
  eq k(canonAttr( (X,Ns) : D ; AtP) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttr( Ns : D ; AtP) ~> canonAttrHolder(AtP' ; X : D) ~> K) .
  eq k(canonAttr( X : D ; AtP) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttr( AtP) ~> canonAttrHolder(AtP' ; X : D) ~> K) .
  eq k(canonAttr( (X,Ns) : D) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttr( Ns : D) ~> canonAttrHolder(AtP' ; X : D) ~> K) .
  eq k(canonAttr( X : D) ~> canonAttrHolder(AtP') ~> K) =
     k(canonAttrHolder(AtP' ; X : D) ~> K) .
  eq k(canonAttrHolder(emptyAtP ; AtP) ~> K) =
     k(attrList(AtP) ~> K) .

***
*** Come up with a list of individual fields -- we can then use this
*** for lists of field names, space allocation, etc.
***
  eq fields(emptyAtP) = emptyNL .
  eq fields( (X : D) ) = X .
  eq fields( (X : D) ; AtP) = X , fields(AtP) .
  eq fields( (X : OD) ) = X .
  eq fields( (X : OD) ; AtP) = X , fields(AtP) .
  eq fields( (X :< Xc) ) = X .
  eq fields( (X :< Xc) ; AtP) = X , fields(AtP) .
  eq fields( (X :< OD) ) = X .
  eq fields( (X :< OD) ; AtP) = X , fields(AtP) .
  eq fields( (X ::< Xc) ) = X .
  eq fields( (X ::< Xc) ; AtP) = X , fields(AtP) .
  eq fields( (X ::< OD) ) = X .
  eq fields( (X ::< OD) ; AtP) = X , fields(AtP) .

endm

mod OO-BETA-HELPING-OPERATIONS is
  including OO-BETA-STATE .
  including BETA-HELPING-OPERATIONS .
  including DECLARATION-SYNTAX .
  including CANONICALIZE-PATTERN .

  sort ClosureList .

  vars Xc Xc' : Name .  var K : Continuation . vars V V' : Value .
  vars Vl Vl' Vl'' : ValueList .  vars O O' : Object .  var OEnv : ObjEnv .
  var N : Nat . var Mem : Store . var TS : PLThreadState . var L : Location .
  vars P P' : PatternItem .

  op _~>_ : ClosureList Continuation -> Continuation .
  op clist : ValueList -> ClosureList .
  eq k(clist(Vl) ~> K) innerList(Vl') = k(K) innerList(Vl) .
  eq k(Vl'' ~> clist(Vl) ~> K) innerList(Vl') = k(Vl'' ~> K) innerList(Vl) .

  op resetClassTo : Value -> ContinuationItem .
  eq k(Vl ~> resetClassTo(V) ~> K) class(V') = k(Vl ~> K) class(V) .
  eq k(resetClassTo(V) ~> K) class(V') = k(K) class(V) .

  op resetObjectTo_ : Object -> ContinuationItem .
  eq k(Vl ~> resetObjectTo O ~> K) obj(O') = k(Vl ~> K) obj(O) .
  eq k(resetObjectTo O ~> K) obj(O') = k(K) obj(O) .

  op allocRefFor_ : Value -> ContinuationItem .
  eq t(k(allocRefFor(V) ~> K) TS) mem(Mem) nextLoc(N)
   = t(k(oref(loc(N)) ~> K) TS) mem(Mem [loc(N),V]) nextLoc(N + 1) .

  op createRepetition : Declaration -> ContinuationItem .
  op createDec : Declaration -> ContinuationItem .

endm

mod GENERIC-EXP-SEMANTICS is
  including OO-BETA-HELPING-OPERATIONS .
  op int : Int -> Value .
  var I : Int .  var X : Name .  var L : Location .  var V : Value .
  var K : Continuation .  var Env : Env .  var Mem : Store . var T : Type .
  var TS : PLThreadState .

  eq k(I ~> K) = k(int(I) ~> K) .
  eq k(emptyImp ~> K) = k(K) .
endm

mod ARITH-IMP-SEMANTICS is
  including ARITH-IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  vars E E' : Exp .  var K : Continuation .  vars I I' : Int .
  ops + - * div % u- : -> ContinuationItem .
  eq k(E + E' ~> K) = k((E,E') ~> + ~> K) .
  eq k((int(I),int(I')) ~> + ~> K) = k(int(I + I') ~> K) .
  eq k(E - E' ~> K) = k((E,E') ~> - ~> K) .
  eq k((int(I),int(I')) ~> - ~> K) = k(int(I - I') ~> K) .
  eq k(E * E' ~> K) = k((E,E') ~> * ~> K) .
  eq k((int(I),int(I')) ~> * ~> K) = k(int(I * I') ~> K) .
  eq k(E div E' ~> K) = k((E,E') ~> div ~> K) .
  eq k((int(I),int(I')) ~> div ~> K) = k(int(I quo I') ~> K) .
***  eq k(E % E' ~> K) = k((E,E') ~> % ~> K) .
***  eq k((int(I),int(I')) ~> % ~> K) = k(int(I rem I') ~> K) .
  eq k((- E) ~> K) = k(E ~> u- ~> K) .
  eq k(int(I) ~> u- ~> K) = k(int(I * (-1)) ~> K) .
endm

mod BOOL-IMP-SEMANTICS is
  including BOOL-IMP-SYNTAX .
  including RELATION-IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  ops eq neq lt leq gt geq and or not : -> ContinuationItem .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .

  eq k(True ~> K) = k(bool(true) ~> K) .
  eq k(False ~> K) = k(bool(false) ~> K) .
  eq k(E eq E' ~> K) = k((E,E') ~> eq ~> K) .
  eq k((int(I),int(I')) ~> eq ~> K) = k(bool(I == I') ~> K) .
  eq k(E neq E' ~> K) = k((E,E') ~> neq ~> K) .
  eq k((int(I),int(I')) ~> neq ~> K) = k(bool(I =/= I') ~> K) .
  eq k(E lt E' ~> K) = k((E,E') ~> lt ~> K) .
  eq k((int(I),int(I')) ~> lt ~> K) = k(bool(I < I') ~> K) .
  eq k(E leq E' ~> K) = k((E,E') ~> leq ~> K) .
  eq k((int(I),int(I')) ~> leq ~> K) = k(bool(I <= I') ~> K) .
  eq k(E gt E' ~> K) = k((E,E') ~> gt ~> K) .
  eq k((int(I),int(I')) ~> gt ~> K) = k(bool(I > I') ~> K) .
  eq k(E geq E' ~> K) = k((E,E') ~> geq ~> K) .
  eq k((int(I),int(I')) ~> geq ~> K) = k(bool(I >= I') ~> K) .
  eq k(E and E' ~> K) = k((E,E') ~> and ~> K) .
  eq k((bool(B),bool(B')) ~> and ~> K) = k(bool(B and B') ~> K) .
  eq k(E or E' ~> K) = k((E,E') ~> or ~> K) .
  eq k((bool(B),bool(B')) ~> or ~> K) = k(bool(B or B') ~> K) .
  eq k(not E ~> K) = k(E ~> not ~> K) .
  eq k(bool(B) ~> not ~> K) = k(bool(not B) ~> K) .
endm

mod FOR-SEMANTICS is
  including FOR-SYNTAX .
  including GENERIC-EXP-SEMANTICS .

  vars E E' : Exp .  var K : Continuation .  var B : Bool .
  vars Is Is' : ImpList . vars V V' : Value . var X : Name .
  vars I I' : Int . vars Env Env' : Env .

  op for : Name ImpList -> ContinuationItem .
  op for2 : Name Int Int ImpList -> ContinuationItem .

  eq k( (for X : E repeat Is for) ~> K) env(Env) =
     k(E ~> for(X,Is) ~> Env ~> K) env(Env) .
  eq k(int(I) ~> for(X,Is) ~> K) =
     k(int(1) ~> bindTo(X) ~> for2(X,1,I,Is) ~> K) .
 ceq k(for2(X,I,I',Is) ~> K) = k(Is ~> discard ~> int(I + 1) ~>
       assignTo(X) ~> for2(X,I + 1,I',Is) ~> K)
     if I <= I' .
  eq k(for2(X,I,I',Is) ~> K) = k(K) [owise] .
endm

mod IF-SEMANTICS is
  including IF-SYNTAX .
  including GENERIC-EXP-SEMANTICS .

  vars E E' : Exp .  var K : Continuation .  var B : Bool .
  var Cs : Cases . vars Is Is' : ImpList . vars V V' : Value .
  vars CS1 CS2 : CaseSelection . vars CSs CSs' : CaseSelections .
  var C : Case . vars Env Env' : Env .


  op if : Cases ImpList -> ContinuationItem .
  op if2 : Value Cases ImpList -> ContinuationItem .
  op if3 : Value ImpList -> ContinuationItem .
  op if4 : Cases -> ContinuationItem .
  op if5 : Value -> ContinuationItem .
  op case : Case -> ContinuationItem .
  op cases : CaseSelections ImpList -> ContinuationItem .
  op cases2 : ImpList -> ContinuationItem .
  op casesel : CaseSelection -> ContinuationItem .
  op skip : -> Exp .

  *** First, top of the if statement
  eq k( (if E Cs else Is if) ~> K) env(Env) = k(E ~> if(Cs,Is) ~> Env ~> K) env(Env) .
  eq k( (if E Cs if) ~> K) env(Env) = k(E ~> if(Cs,skip) ~> Env ~> K) env(Env) .

  *** Since we converted a statement with no else to one with an
  *** else that does nothing, handle the "does nothing" case
  eq k(skip ~> K) = k(K) .

  *** Next, after evaluating top value, set up cases
  eq k(V ~> if((C Cs),Is) ~> K) = k(case(C) ~> if2(V,Cs,Is) ~> K) .
  eq k(V ~> if(C,Is) ~> K) = k(case(C) ~> if3(V,Is) ~> K) .

  *** Break the cases apart into individual cases to check
  eq k(case((CS1 CSs) then Is') ~> K) = k(casesel(CS1) ~> cases(CSs,Is') ~> K).
  eq k(case((CS1) then Is') ~> K) = k(casesel(CS1) ~> cases2(Is') ~> K).

  *** Evaluate an individual case selector
  eq k(casesel(// E) ~> K) = k(E ~> K) .

  *** Match against returned case value to see if we have a hit
  eq k(V ~> cases(CSs,Is') ~> if2(V,Cs,Is) ~>  K) = k(Is' ~> K) .
  eq k(V ~> cases(CSs,Is') ~> if2(V',Cs,Is) ~> K) =
     k(case((CSs) then Is') ~> if2(V',Cs,Is) ~> K) .
  eq k(V ~> cases(CSs,Is') ~> if3(V,Is) ~>  K) = k(Is' ~> K) .
  eq k(V ~> cases(CSs,Is') ~> if3(V',Is) ~> K) =
     k(case((CSs) then Is') ~> if3(V',Is) ~> K) .
  eq k(V ~> cases2(Is') ~> if2(V,Cs,Is) ~>  K) = k(Is' ~> K) .
  eq k(V ~> cases2(Is') ~> if2(V',Cs,Is) ~> K) = k(V' ~> if(Cs,Is) ~> K) .
  eq k(V ~> cases2(Is') ~> if3(V,Is) ~>  K) = k(Is' ~> K) .
  eq k(V ~> cases2(Is') ~> if3(V',Is) ~> K) = k(Is ~> K) .
endm

mod SEQ-COMP-SEMANTICS is
  including IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  vars I I' : Imp .  var Is : ImpList . var K : Continuation .
  eq k((I ; Is) ~> K) = k(I ~> discard ~> Is ~> K) .
  eq k(discard ~> K) = k(K) . *** don't leave discard on top
endm

mod LABELED-IMP-SEMANTICS is
  including CONTROL-IMP-SYNTAX .
  including SEQ-COMP-SEMANTICS .

  vars I I' : Imp . vars Is Is' : ImpList . var K : Continuation .
  vars X X' : Name . var CI : ContinuationItem .
  vars Env Env' : Env .

  op label : Name ImpList Env -> ContinuationItem .
  op leaving : Name -> ContinuationItem .
  op restarting : Name -> ContinuationItem .
```

```
***
*** When we find a labeled ImpList, put a label token
*** in the continuation so we can find it later if we
*** need it.
***
  eq k( ((X : Is : X)) ~> K) env(Env) =
     k(Is ~> label(X,Is,Env) ~> K) env(Env) .

***
*** If we hit the label normally later, just throw it
*** away .
***
  eq k(label(X,Is,Env) ~> K) = k(K) .

***
*** If we have a leave x, set leaving so we can throw
*** away continuation items until we find the proper label.
***
  eq k(leave X ~> K) = k(leaving(X) ~> K) .

***
*** If we find the associated label, just start going again.
***
  eq k(leaving(X) ~> label(X,Is,Env) ~> K) env(Env') = k(K) env(Env) .

***
*** If we find anything else, discard it and keep looking.
***
  eq k(leaving(X) ~> CI ~> K) env(Env) = k(leaving(X) ~> K) env(Env) [owise] .
  eq k(leaving(X) ~> Env' ~> K) env(Env) = k(leaving(X) ~> K) env(Env) [owise] .
  eq k(leaving(X) ~> CL::ClosureList ~> K) env(Env) =
     k(leaving(X) ~> K) env(Env) [owise] .
  eq k(leaving(X) ~> NL::NameList ~> K) env(Env) =
     k(leaving(X) ~> K) env(Env) [owise] .
  eq k(leaving(X) ~> EL::ExpList ~> K) env(Env) =
     k(leaving(X) ~> K) env(Env) [owise] .
  eq k(leaving(X) ~> IL::ImpList ~> K) env(Env) =
     k(leaving(X) ~> K) env(Env) [owise] .

***
*** If we have a restart x, set restarting so we can
*** throw away continuation items until we find the
*** proper label.
***
  eq k(restart X ~> K) = k(restarting(X) ~> K) .

***
*** If we find the associated label, restart the ImpList associated
*** with this label.
***
```

```
  eq k(restarting(X) ~> label(X,Is,Env) ~> K) env(Env') =
     k(Is ~> label(X,Is,Env) ~> K) env(Env) .

***
*** If we find anything else, discard it and keep looking.
***
  eq k(restarting(X) ~> CI ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
  eq k(restarting(X) ~> Env' ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
  eq k(restarting(X) ~> CL::ClosureList ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
  eq k(restarting(X) ~> NL::NameList ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
  eq k(restarting(X) ~> EL::ExpList ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
  eq k(restarting(X) ~> IL::ImpList ~> K) env(Env) =
     k(restarting(X) ~> K) env(Env) [owise] .
endm

***
*** Look up names in the current environment. Name lookup first
*** checks in the environment at the top level, and then looks
*** at the environment in each layer of the current instantiated
*** object.
***
*** TODO : We need to modify this to force only names in the
*** pattern to be used once we look at objects, since we could
*** look up a name in scope but outside of the pattern. We can
*** either modify the dynamic semantics here or (preferably)
*** check this in a static checker.
***
mod LOOKUP-SEMANTICS is
  including GENERIC-EXP-SEMANTICS .
  including CALL-SYNTAX .

  op deref : Value -> Value .
  op lookup : Name -> ContinuationItem .
  op lookupName : Name Object -> ContinuationItem .
  op lookupForWrite : Name Value Object -> ContinuationItem .
  op lookupLoc : Name -> ContinuationItem .
  op lookupNameLoc : Name Object -> ContinuationItem .
  op loc : Location -> Value .
  op {_|_,_} : Name Name Object -> ContinuationItem .
  op {|_|_,_|} : Name Name Object -> ContinuationItem .
  op semaphorRead : Name -> ContinuationItem .

  var I : Int .  var V : Value .  var T : Type .
  var OD : ObjDesc . var K : Continuation .
  vars Ns Ns' : NameList . var AtP : AttributePart . var Is : ImpList .
  var N : Nat . vars X Xc Xc' Xc'' : Name . vars Env Env' : Env .
  var Vl : ValueList .  vars P P' : Pattern .
  vars O O' : Object .  vars L L' TL : Location .
  var OEnv : ObjEnv .  var Mem : Store .  var Xl : NameList .
  var TS : PLThreadState . var E : Exp .

  rl t(k(X ~> K) env([X,L] Env) TS) mem([L,V] Mem)
  => t(k(V ~> K) env([X,L] Env) TS) mem([L,V] Mem) .
  rl t(k(deref(oref(L)) ~> K) TS) mem([L,V] Mem)
  => t(k(V ~> K) TS) mem([L,V] Mem) .
  rl t(k(lookupLoc(X) ~> K) env([X,L] Env) TS) mem([L,V] Mem)
  => t(k(loc(L) ~> K) env([X,L] Env) TS) mem([L,V] Mem) .

*** A Semaphor should not be available for concurrent access.
*** Hence this is not a rl.
  eq t(k(semaphorRead(X) ~> K) env([X,L] Env) TS) mem([L,V] Mem)
   = t(k((loc(L),V) ~> K) env([X,L] Env) TS) mem([L,V] Mem) .

***
*** Look up names within this object. We need to switch environments
*** when we switch names to ensure we find the correctly scoped parent.
***
  crl k(X ~> K) class(pt(P pname(Xc))) obj(O) env(Env)
  => k({X | Xc, O} ~> Env ~> K) obj(O) class(pt(P pname(Xc))) env(Env)
  if ( X %% Env ) =/= true .

  rl t(k({X | Xc, o(Xc', L', [Xc, [X,L] Env] OEnv)} ~> K) TS) mem([L,V] Mem)
  => t(k(V ~> K) TS) mem([L,V] Mem) .

  crl t(k({X | Xc, o(Xc',L,OEnv)} ~> K) TS) mem(Mem)
  => t(k(Xc ~> lookupName(X,o(Xc',L,OEnv)) ~> K) TS) mem(Mem)
  if (X % Xc %% OEnv) =/= true .

*** Typing rules should prevent this situation from happening,
*** but we are running somewhat dynamically right now and need
*** this to stop recursion.
  eq t(k(pt(parent(HierTop) P) ~> lookupName(X,O) ~> K)
     env(Env') TS) mem(Mem)
   = t(k(nothing ~> K) env(Env') TS) mem(Mem) .

  eq t(k(pt(parent(Xc) P) ~> lookupName(X,o(Xc',L,[Xc,Env] OEnv)) ~> K)
     env(Env') TS) mem(Mem)
   = t(k({X | Xc, o(Xc',L,[Xc,Env] OEnv)} ~> K) env(Env) TS) mem(Mem) [owise] .

***
*** Equivalent rules to the above, but instead for looking up the location
*** of a given name.
***
***
*** Look up names within this object. We need to switch environments when we
*** switch names to ensure we find the correctly scoped parent.
***
  crl k(lookupLoc(X) ~> K) class(pt(P pname(Xc))) obj(O) env(Env)
  => k({| X | Xc, O |} ~> Env ~> K) obj(O) class(pt(P pname(Xc))) env(Env)
  if ( X %% Env ) =/= true .

  rl t(k({| X | Xc, o(Xc', L', [Xc, [X,L] Env] OEnv) |} ~> K) TS) mem([L,V] Mem)
  => t(k(loc(L) ~> K) TS) mem([L,V] Mem) .

  crl t(k({| X | Xc, o(Xc',L,OEnv) |} ~> K) TS) mem(Mem)
  => t(k(Xc ~> lookupName(X,o(Xc',L,OEnv)) ~> K) TS) mem(Mem)
  if (X % Xc %% OEnv) =/= true .

*** Typing rules should prevent this situation from happening, but we are running somewhat
*** dynamically right now and need this to stop recursion.
  eq t(k(pt(parent(HierTop) P) ~> lookupName(X,O) ~> K) env(Env') TS) mem(Mem)
   = t(k(nothing ~> K) env(Env') TS) mem(Mem) .

  eq t(k(pt(parent(Xc) P) ~> lookupName(X,o(Xc',L,[Xc,Env] OEnv)) ~> K) env(Env') TS) mem(Mem)
   = t(k({| X | Xc, o(Xc',L,[Xc,Env] OEnv) |} ~> K) env(Env) TS) mem(Mem) [owise] .

***
*** Lookup remote names
***
  eq k((E . X) ~> K) env(Env) = k(E ~> lookup(X) ~> Env ~> K) env(Env) .
  eq k(oref(L) ~> lookup(X) ~> K) = k(deref(oref(L)) ~> lookup(X) ~> K) .
  rl t(k(o(Xc, L, [Xc,Env] OEnv) ~> lookup(X) ~> K) obj(O')
       class(pt(P)) env(Env') TS) mem(Mem [L,pt(P')]) =>
     t(k({X | Xc, o(Xc, L, [Xc,Env] OEnv)} ~> resetClassTo(pt(P)) ~>
       resetObjectTo O' ~> Env' ~> K) obj(o(Xc, L, [Xc,Env] OEnv))
       class(pt(P')) env(Env) TS) mem(Mem [L,pt(P')]) .

***
*** Lookup remote names for assignment
***
 op lookupForAssign : Exp -> ContinuationItem .
 eq k(lookupForAssign(E . X) ~> K) env(Env) = k(E ~> lookupLoc(X) ~> Env ~> K) env(Env) .
 eq k(oref(L) ~> lookupLoc(X) ~> K) = k(deref(oref(L)) ~> lookupLoc(X) ~> K) .
  rl t(k(o(Xc, L, [Xc,Env] OEnv) ~> lookupLoc(X) ~> K) obj(O')
       class(pt(P)) env(Env') TS) mem(Mem [L,pt(P')]) =>
     t(k({| X | Xc, o(Xc, L, [Xc,Env] OEnv) |} ~> resetClassTo(pt(P)) ~>
       resetObjectTo O' ~> Env' ~> K) obj(o(Xc, L, [Xc,Env] OEnv))
       class(pt(P')) env(Env) TS) mem(Mem [L,pt(P')]) .

***
*** Specify lookup semantics for virtual methods -- lookup should go here, as this
*** should be of no concern to the actual semantics of invoking the pattern once
*** we have it.
***
*** NOTE: The block item just wraps a value to keep from collapsing two adjacent
*** values into a value list.
***
  op block : Value -> ContinuationItem .
  op dispatch : Type Location Value Value -> ContinuationItem .
  eq k(href(T,L,oref(L')) ~> lookup(X) ~> K) =
     k(ldir(L) ~> block(href(T,L,oref(L'))) ~> lookup(X) ~> K) .
  eq k(pt(pname(Xc) P) ~> block(href(T,L,oref(L'))) ~> lookup(X) ~> K) =
     k(deref(oref(L')) ~> dispatch(T,L,oref(L'),pt(pname(Xc) P)) ~> lookup(X) ~> K) .
  eq k(o(Xc, L, [Xc',Env] OEnv) ~> dispatch(T,TL,oref(L'),pt(pname(Xc') P)) ~>
       lookup(X) ~> K)
     obj(O') class(pt(P')) env(Env') =
     k({X | Xc', o(Xc, L, [Xc',Env] OEnv)} ~> resetClassTo(pt(P')) ~>
     resetObjectTo O' ~> Env' ~> dispatch(T,TL,oref(L'),pt(pname(Xc') P)) ~>
     lookup(X) ~> K)
     obj(o(Xc, L, [Xc',Env] OEnv)) class(pt(pname(Xc') P)) env(Env) .
*** If the pattern that comes back is marked virtual, dispatch to top
  eq k(pt(virtual(true) P) ~> dispatch(T,TL,oref(L'),pt(P')) ~> lookup(X) ~> K) =
     k(oref(L') ~> lookup(X) ~> K) .
*** If not, dispatch at this level
  eq k(pt(virtual(false) P) ~> dispatch(T,TL,oref(L'),pt(P')) ~> lookup(X) ~> K) =
     k(pt(virtual(false) P) ~> K) .

endm

***
*** Semantics for non-reference assignments, such as
*** 5 -> a, where a is defined as a : @integer
***
mod ASSIGNMENT-SEMANTICS is
  including GENERAL-IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including LOOKUP-SEMANTICS .

  var E : Exp .  var El : ExpList .
  var I : Int .  var V : Value .  var T : Type .
  var OD : ObjDesc . var K : Continuation .
  vars Ns Ns' : NameList . var AtP : AttributePart . var Is : ImpList .
  var N : Nat . vars X Xc Xc' : Name . vars Env Env' : Env .
  var Vl : ValueList .  vars P P' P'' : Pattern .
  vars O O' : Object .  vars L L' : Location .
  var OEnv : ObjEnv .  var Mem : Store .  var Xl : NameList .
  var TS : PLThreadState .

  eq k((E -> X) ~> K) = k(E ~> assignTo(X) ~> X ~> K) .
  eq k((El -> Xl) ~> K) = k(El ~> assignTo(Xl) ~> Xl ~> K) .

***
*** Assign to names within this object. See above for why we switch environments.
***
  op {_<-_|_,_} : Name Value Name Object -> ContinuationItem .
  crl t(k((V,Vl) ~> assignTo(X,Xl) ~> K) class(pt(pname(Xc) P)) obj(O) env(Env) TS)
  => t(k({X <- V | Xc, O} ~> Env ~> Vl ~> assignTo(Xl) ~> K) class(pt(pname(Xc) P))
    obj(O) env(Env) TS)
  if (X %% Env) =/= true .

  crl t(k(V ~> assignTo(X) ~> K) class(pt(pname(Xc) P)) obj(O) env(Env) TS)
  => t(k({X <- V | Xc, O} ~> Env ~> K) class(pt(pname(Xc) P)) obj(O) env(Env) TS)
  if (X %% Env) =/= true .

  rl t(k({X <- V | Xc', o(Xc, L', [Xc', [X,L] Env] OEnv)} ~> K) TS) mem(Mem)
  => t(k(K) TS) mem(Mem[L <- V]) .

  crl t(k({X <- V | Xc, o(Xc', L', OEnv)} ~> K) TS) mem(Mem)
  => t(k(Xc ~> lookupForWrite(X,V,o(Xc', L', OEnv)) ~> K) TS) mem(Mem)
  if (X % Xc %% OEnv) =/= true .

*** Typing rules should prevent this situation from happening,
*** but we are running somewhat dynamically right now and need this
*** to stop recursion.
  eq t(k(pt(parent(HierTop) P) ~> lookupForWrite(X,V,O) ~> K) env(Env') TS) mem(Mem)
   = t(k(nothing ~> K) env(Env') TS) mem(Mem)  .

  eq t(k(pt(parent(Xc) P) ~> lookupForWrite(X,V,o(Xc',L,[Xc,Env] OEnv)) ~> K)
       env(Env') TS) mem(Mem)
   = t(k({X <- V | Xc, o(Xc',L,[Xc,Env] OEnv)} ~> K) env(Env) TS) mem(Mem) [owise] .

endm

mod REPETITION-SEMANTICS is
  including REPETITION-IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including OO-BETA-HELPING-OPERATIONS .
  including SLICE-SEMANTICS .
  op rep : ValueList Declaration Int -> Value .
  op createRepetition : Declaration Int ValueList Int -> ContinuationItem .
  op rep : Exp -> ContinuationItem .
  op getArrayElements : Int Int -> ContinuationItem .
  vars I I' N : Int . var Dec : Declaration . var K : Continuation .
  vars VL VL' : ValueList . vars V V' : Value . vars E E' Re : Exp .
  var X : Name . vars El El' : ExpList .

  *** creating a repetition
  eq k(int(I) ~> createRepetition(Dec) ~> K) = k(createRepetition(Dec, I, nil, 0) ~> K) .
  eq k(createRepetition(Dec, I, VL, I) ~> K) = k(rep(VL, Dec, I) ~> K) .
 ceq k(createRepetition(Dec, I, VL, I') ~> K) = k(createDec(Dec) ~>
      createRepetition(Dec, I, VL, (I' + 1)) ~> K)
   if I > I' .
  eq k(V ~> createRepetition(Dec, I, VL, I') ~> K) =
     k(createRepetition(Dec, I, (VL, V), I') ~> K) .
  *** accessing element(s) of a repetition
  eq k(Re[E] ~> K) = k(E ~> rep(Re) ~> K) .
  eq k(Re[ E $ E' ] ~> K) = k((E, E') ~> rep(Re) ~> K) .
  eq k(int(I) ~> rep(Re) ~> K) = k(Re ~> getArrayElements(I, I) ~> K) .
 ceq k((int(I),int(I')) ~> rep(Re) ~> K) = k(Re ~> getArrayElements(I, I') ~> K) if I <= I' .
  eq k(rep(VL, Dec, N) ~> getArrayElements(I, I') ~> K) = k(getSlice(VL, I, I') ~> K) .

  *** assignment to repetition elements
  ops assignArray assignArraySlice : -> ContinuationItem .
  eq k((E -> X[E']) ~> K) = k((E, E', X) ~> assignArray ~> assignTo(X) ~> K) .
  eq k((V, int(I), rep(VL, Dec, N)) ~> assignArray ~> K) =
     k(rep(setSlice(VL, I, I, V), Dec, N) ~> K) .
  eq k((El -> X[E $ E']) ~> K) =
     k((El,E,E',X) ~> assignArraySlice ~> assignTo(X) ~> K) .
 ceq k((VL', int(I), int(I'), rep(VL, Dec, N)) ~> assignArraySlice ~> K) =
     k(rep(setSlice(VL, I, I', VL'), Dec, N) ~> K)
   if I <= I' .

  *** range
  op range : -> ContinuationItem .
  eq k(Re .range ~> K) = k(Re ~> range ~> K) .
  eq k(rep(VL, Dec, N) ~> range ~> K) = k(int(N) ~> K) .

  *** extend
  op extendArray : -> ContinuationItem .
  op extendArray : Value -> ContinuationItem .
  eq k((E -> X .extend) ~> K) = k((E, X) ~> extendArray ~> assignTo(X) ~> K) .
  eq k((int(I), rep(VL, Dec, N)) ~> extendArray ~> K) =
     k(int(I) ~> createRepetition(Dec) ~> extendArray(rep(VL, Dec, N)) ~> K) .
  eq k(rep(VL, Dec, I) ~> extendArray(rep(VL', Dec, N)) ~> K) =
     k(rep((VL, VL'), Dec, (I + N)) ~> K) .

  *** new
  op newArray : -> ContinuationItem .
  eq k((E -> X .new) ~> K) = k((E, X) ~> newArray ~> assignTo(X) ~> K) .
  eq k((int(I), rep(VL, Dec, N)) ~> newArray ~> K) = k(int(I) ~> createRepetition(Dec) ~> K) .

endm

mod PATTERN-MEMBERSHIP-SEMANTICS is
  including PATTERN-MEMBERSHIP-IMP-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including LOOKUP-SEMANTICS .

  var E : Exp . var K : Continuation . vars Xc0 Xc : Name .
  var Env : Env . var L : Location . var T : Type .
  var OEnv : ObjEnv . var P : Pattern .

  op getPattern : -> ContinuationItem .
  eq k((E ##) ~> K) = k(E ~> getPattern ~> K) .
  eq k(oref(L) ~> getPattern ~> K) = k(deref(oref(L)) ~> getPattern ~> K) .
  eq k(o(Xc, L, OEnv) ~> getPattern ~> K) = k(ldir(L) ~> K) .
  eq k(pt(P) ~> getPattern ~> K) = k(pt(P) ~> K) .
endm

***
*** Generalize the method call semantics. Method calls will now work
*** in the following way:
*** 1) First, grab back the reference to the pattern that we are
***    invoking.
*** 2) Second, evaluate parameters, if we have any.
*** 3) Third, invoke the method.
*** 4) Finally, put the result of the method on the continuation,
***    which may then be used or discarded.
***
*** TODO: inner calls with modifiers
***
mod METHOD-SEMANTICS is
  including OO-BETA-HELPING-OPERATIONS .
  including GENERIC-EXP-SEMANTICS .
  including CALL-SYNTAX .
  including LOOKUP-SEMANTICS .

  var K : Continuation .
  vars Env Env' : Env .  var Xl : NameList .  vars E E' : Exp .
  vars Xc Xc' Xm Xc'' XcO XT X VP : Name . var V : Value . vars El El' : ExpList .
  vars Vl Vl' Vl'' : ValueList . var OEnv : ObjEnv . vars O O' : Object .
  vars Ns Ns' : NameList . var AtP : AttributePart . var Is : ImpList .
  var L : Location . var I : Imp . var Dec : Declaration . var OD : ObjDesc .
  vars P P' P'' : Pattern . var Mem : Store . var TS : PLThreadState .

  op source : ValueList -> ContinuationItem .
  op target : ExpList -> ContinuationItem .
  op invokable : Value -> ContinuationItem .
  op referenceable : Value -> ContinuationItem .
  op invoke : Value ValueList -> ContinuationItem .
  op invokeWObj : Value ValueList Object -> ContinuationItem .
  op getClosures : ValueList -> ContinuationItem .
  op closures : ValueList -> ContinuationItem .
  op startAtTop : -> ContinuationItem .

*** This is the general form of an assignment, which can have multiple
*** levels (such as El -> El' -> El'' -> etc...). We will handle the
*** levels from left to right. This is an owise since we have two
*** other competing rules -- value to name assignment, and reference
*** to reference var assignment, and this should only fire if those
*** do not apply.
  eq k((El -> E) ~> K) = k(El ~> target(E) ~> K) [owise] .

*** When we eval and get values back, take the first expression
*** out of the target to get the actual target back and stash
*** the parameters for later use. Check to see if we are assigning
*** or invoking here -- if we are invoking, we want a pattern back,
*** if we are assigning we want the location to assign into back
*** instead (and we aren't creating anything new). For now, assume
*** that we just have a single value on assignments to remote names.
*** We should remove this assumption later.
***
  eq k(Vl ~> target(& E) ~> K) = k((& E) ~> source(Vl) ~> K) .
  eq k(Vl ~> target(E . X) ~> K) = k(lookupForAssign(E . X) ~> source(Vl) ~> K) [owise] .
  eq k(Vl ~> target(E) ~> K) = k((E) ~> source(Vl) ~> K) [owise] .
  eq k(loc(L) ~> source(V) ~> K) = k(lassign(L,V) ~> V ~> K) .

*** When we encounter a pattern creation, create a new object
*** based on the pattern and mark it invokable or referenceable.
*** Don't invoke it here, since we don't check to see if it has
*** params or not at this step. If it is referenceable, we don't
*** want to return the exit values, but a reference to the object
*** instead.
*** TODO: See if we can remove the referenceable handler -- may
*** not be able to remove this special case, since we normally
*** don't return the object, just the return values from exit.
***
  op newObject : Value -> ContinuationItem .
  op lookupForInvoke : -> ContinuationItem .
  op lookupForRef : -> ContinuationItem .
  op _~>_ : PatternDecl Continuation -> Continuation .

  eq k((& E) ~> K) = k(E ~> lookupForInvoke ~> K) .
  eq k(pt(P) ~> lookupForInvoke ~> K) =
     k(newObject(pt(P)) ~> invokable(pt(P)) ~> K) .

  eq k((& E []) ~> K) = k(E ~> lookupForRef ~> K) .
  eq k(pt(P) ~> lookupForRef ~> K) =
     k(newObject(pt(P)) ~> referenceable(pt(P)) ~> K) .

*** Now, we have a new object, which is an object reference. Invoke
*** the object and pass it any parameters it may need. If we don't
*** have parameters, normalize it so we just have an empty param list.
  eq k(oref(L) ~> invokable(pt(P)) ~> source(Vl) ~> K) =
     k(deref(oref(L)) ~> invoke(pt(P),Vl) ~> K) .
  eq k(oref(L) ~> invokable(pt(P)) ~> target(El) ~> source(Vl) ~> K) =
     k(deref(oref(L)) ~> invoke(pt(P),Vl) ~> target(El) ~> K) .
  eq k(oref(L) ~> invokable(pt(P)) ~> K) =
     k(oref(L) ~> invokable(pt(P)) ~> source(nil) ~> K) [owise] .

*** If we have a new object that is instead referenceable, check
*** to see if we have parameters. If we do, we need to invoke
*** a constructor. If not, we just pass the reference back.
*** TODO: Get constructors working. For now, just pass ref back.
```

```
  eq k(oref(L) ~> referenceable(pt(P)) ~> source(Vl) ~> K) =
     k(oref(L) ~> K) .
  eq k(oref(L) ~> referenceable(pt(P)) ~> K) =
     k(oref(L) ~> K) [owise] .

*** This builds the list of calls back to the top, so we
*** can start at the top and then use inner to get back down .
*** NOTE: It may make sense to calculate this statically later,
*** but we currently do it dynamically, which is slower.
*** NOTE: We include an unused attribute below in each of the
*** two equations just to make sure Maude knows they are a pair
*** and will thus correctly use the owise.
 crl t(k(pt(parent(Xc) parentLoc(L) P) ~> getClosures(Vl) ~> K) TS) =>
     t(k(closures(pt(parent(HierTop) parentLoc(L) P), Vl) ~> K) TS)
     if Xc == HierTop .
 crl t(k(pt(parent(Xc) parentLoc(L) P) ~> getClosures(Vl) ~> K) TS) mem([L,V] Mem) =>
     t(k(V ~> getClosures(pt(parent(Xc) parentLoc(L) P), Vl) ~> K) TS) mem([L,V] Mem)
     if Xc =/= HierTop .

*** With the proper object dereferenced and the (potentially
*** empty) parameter list, first build the closure list, which
*** will give us the proper environment for executing the method
*** since we need to start at the top and then use inner to get
*** back down.
  eq k(O ~> invoke(pt(P),Vl) ~> K) =
     k(pt(P) ~> getClosures(nil) ~> invokeWObj(pt(P),Vl,O) ~> K) .

*** Set up the inner call list, now that we have a proper list of
*** closures. This will allow us to call the method in the correct
*** environment. We also want to make sure we replace this with
*** whatever is current when the call ends, so we don't inadvertently
*** switch to the wrong environment.
  eq k(closures(Vl,V) ~> invokeWObj(pt(P),Vl'',O) ~> K) innerList(Vl') =
     k(V ~> invokeWObj(pt(P),Vl'',O) ~> clist(Vl') ~> K) innerList(Vl,V) .

*** Now, finally, we can execute the code in the pattern, since we
*** have the proper starting pattern. Assign any params, run the do
*** code, and then return any exit block values. We also need to take
*** care of the class and object environments here, to make sure they
*** are properly set/reset.
*** TODO: We are assuming at this point that the method signature does
*** not change, but this is not actually true in Beta. The method signature
*** is equivalently the concatenation of all given signatures through the
*** levels, with params in scope only at the level they are assigned. So,
*** method foo can have entry a, then a child method foo can have entry b,c,
*** with the method expecting (a,b,c) -> as input to the parameter.
  eq k(pt(entryList(Ns) exitList(Ns') pEnv(Env) P) ~>
     invokeWObj(pt(pname(Xc) P'),Vl,O) ~> K)
     obj(O') class(pt(pname(Xc'') P'')) env(Env')
   = k(Vl ~> assignTo Ns ~> startAtTop ~> discard ~> Ns' ~> resetObjectTo(O') ~>
     resetClassTo(pt(pname(Xc'') P'')) ~> Env' ~> K)
     obj(O) class(pt(pname(Xc) P')) env(Env) [owise] .

***
*** A couple of handy equations to remove special cases above. The first
*** handles the case where we assign an empty value list to an empty name
*** list, while the second handles the case where we return an empty name
*** list on top of the continuation.
***
  eq k((nil).ValueList ~> assignTo emptyNL ~> K) = k(K) .
  eq k(emptyNL ~> K) = k(K) .

*** To match the inner semantics of Beta, we start at the top of the inner
*** list with with startAtTop call. This will just pick out the do code for
*** the first (which will actually always be Object, which just has code
*** inner), allowing us to work our way down.
  var VC : Value .
  eq k(startAtTop ~> K) class(VC) innerList(pt(doCode(Is) pEnv(Env) P), Vl) env(Env')
   = k(Is ~> Env' ~> resetClassTo(VC) ~> K) class(pt(doCode(Is) pEnv(Env) P))
     innerList(Vl) env(Env) .

*** Now, handle inner. We just assume for now that we have inner with
*** no pattern name. We don't need to ability to inner from an enclosing
*** do block at this time, but should add this in the future.
  eq k(inner ~> K) innerList(pt(doCode(Is) pEnv(Env) P),Vl) class(VC) env(Env')
   = k(Is ~> clist(pt(doCode(Is) pEnv(Env) P),Vl) ~> Env' ~> resetClassTo(VC) ~> K)
     class(pt(doCode(Is) pEnv(Env) P)) innerList(Vl) env(Env) .
  eq k(inner ~> K) innerList(nil) class(VC) env(Env')
   = k(K) innerList(nil) class(VC) env(Env') .

*** Throw away values that get "stuck" on top of an object ref.
  eq k(Vl ~> oref(L) ~> K) = k(oref(L) ~> K) .

endm

***
*** Model how references work, including creating references
*** to new objects and assigning references to
*** existing objects.
***
*** Core operations supported:
*** &Pattern[]
*** reference -> Var[]
*** NONE -> Var[]
***
mod REFERENCE-SEMANTICS is
  including METHOD-SEMANTICS .

  var K : Continuation . var E : Exp . var X : Name .
```

```
*** assign the reference to a reference cell (cannot be
*** defined as @something, which is a stack allocation)
  eq k((E -> (X [])) ~> K) = k((E,X) ~> assignToRef(X) ~> nothing ~> K) .
endm

***
*** We will set up a class as a collection of its important parts -- declarations,
***   entry block, code block, exit block.
***
mod CLASS-SEMANTICS is
  including ATTRIBUTE-SYNTAX .
  including ATTRIBUTE-PART-SYNTAX .
  including ACTION-PART-SYNTAX .
  including OBJECT-DESCRIPTOR-SYNTAX .
  including DECLARATION-SYNTAX .
  including BETA-TYPES .
  including BETA-HELPING-OPERATIONS .
  including OO-BETA-HELPING-OPERATIONS .
  including METHOD-SEMANTICS .
  including REFERENCE-SEMANTICS .
  including LOOKUP-SEMANTICS .

  var OD : ObjDesc . var K : Continuation .
  vars Ns Ns' : NameList . var AtP : AttributePart . var Is : ImpList .
  var N : Nat . vars X Xc Xc' : Name . vars Env Env' : Env .
  var Vl : ValueList .  var V : Value . var P : Pattern .
  vars O O' : Object .  vars L L' : Location .
  var OEnv : ObjEnv .  var Mem : Store .  var Xl : NameList .
  var TS : PLThreadState . var E : Exp .

  op _~>_ : ObjDesc Continuation -> Continuation .
  op canonHolder : Name ObjDesc Env -> ContinuationItem .

***
*** We need to first allocate space for the pattern, so we can track that
*** in the pattern itself. This lets us use the location of the pattern
*** in the store for pattern uniqueness, since it is possible to have multiple
*** patterns that have the same name.
***
  eq t(k(OD ~> K) env(Env') class(pt(pname(Xc) P)) obj(o(Xc',L,[Xc,Env] OEnv)) TS) nextAnon(N)
   = t(k(bindToNothingWLoc(anonObj(N)) ~> canonHolder(anonObj(N),OD,Env) ~> K)
     env(Env') class(pt(pname(Xc) P)) obj(o(Xc',L,[Xc,Env] OEnv)) TS) nextAnon(N + 1) .
  eq k((X : OD) ~> K) env(Env') class(pt(pname(Xc) P)) obj(o(Xc',L,[Xc,Env] OEnv))
   = k(bindToNothingWLoc(X) ~> canonHolder(X,OD,Env) ~> K) env(Env')
     class(pt(pname(Xc) P)) obj(o(Xc',L,[Xc,Env] OEnv)) .

  eq k(lref(L) ~> canonHolder(X,OD,Env) ~> K) =
     k(canonObj(X,OD,(Env [X,L]),L) ~> assignTo(X) ~> K) .

***
```

```
*** Handle inserted objects (we should move this out at some point, but this
*** is the most similar code to what we are doing now)
***
 eq t(k(ins(OD) ~> K) TS) nextAnon(N) =
    t(k((anonObj(N) : OD) ~> & anonObj(N) ~> K) TS) nextAnon(N + 1) .
 eq k(ins(E) ~> K) = k(& E ~> K) .

endm


***
*** Provide semantics for the creation of a new object. This should handle
*** creating the internal structure of the object and should then hand
*** back a reference to the created object. It should invoke nothing --
*** this is handled instead by the method semantics, which handle invoking
*** all code within a pattern.
***
mod NEW-SEMANTICS is
  including CLASS-SEMANTICS .
  including GENERIC-EXP-SEMANTICS .
  including METHOD-SEMANTICS .

  var OD : ObjDesc . var K : Continuation .
  vars Ns Ns' : NameList . var AtP : AttributePart . var Is : ImpList .
  var OM : ObjMain . var Dec : Declaration .
  vars Xc Xc' Xc'' XcO : Name .  var El : ExpList .  var O : Object .
  vars Env Env' : Env .  var OEnv : ObjEnv . var X : Name . var T : Type .
  var V : Value . vars P P' : Pattern . vars L L' : Location .
  var E : Exp . var Mem : Store . var TS : PLThreadState .

  op finalizeObject  : Value -> ContinuationItem .
  op executeDo : -> ContinuationItem .
  op createObject : Value -> ContinuationItem .
  op returnObjectRef : -> ContinuationItem .

  ***
  *** For a new object request, we need to look up the pattern given by
  *** the name, create the object layout, allocate a reference for it,
  *** and then reset the environment, just in case we changed it.
  ***
  eq k(newObject(pt(P)) ~> K) env(Env) =
     k(pt(P) ~> createObject(pt(P)) ~> finalizeObject(pt(P)) ~>
       returnObjectRef ~> Env ~> K) env(Env) .

  ***
  *** Build the structure of the object by recursively building each parent
  *** until we get to the top of the hierarchy
  ***
 crl t(k(pt(parent(Xc) parentLoc(L) P) ~> createObject(pt(P')) ~>
        finalizeObject(pt(P')) ~> K) TS) mem(Mem) =>
     t(k(o(Object, loc(0), [Object,[Object,loc(0)]]) ~> K) TS) mem(Mem)
     if Xc == HierTop .
 crl t(k(pt(parent(Xc) parentLoc(L) P) ~> createObject(pt(P')) ~> K) TS) mem(Mem [L,V]) =>
     t(k(V ~> createObject(V) ~> finalizeObject(V) ~> K) TS)
     mem(Mem [L,V])
     if Xc =/= HierTop .

  ***
  *** Now that all the lower levels have been allocated, allocate storage for this class and
  *** add a layer for this class with the proper environment. Note that we bind and then
  *** allocate to make sure any nested patterns can see all the names in this scope. We also
  *** skip the object around so we can use it in the addLayer, since the object represents
  *** the level we are building on top of.
  ***
  eq k(o(X, L, OEnv) ~> finalizeObject(pt(attributes(AtP) pEnv(Env) P)) ~> K) env(Env') =
     k(bindToNothing(fields(AtP)) ~> allocFields(AtP) ~> o(X, L, OEnv) ~>
       addLayerFor(pt(attributes(AtP) pEnv(Env) P)) ~> K) env(Env) .

  ***
  *** Take the current object and add a new layer for the new class. This lets us build up layers
  *** for subclasses and maintain the environments for superclasses properly.
  ***
  op addLayerFor : Value -> ContinuationItem .
  eq k(o(Xc', L', OEnv) ~> addLayerFor(pt(pname(Xc) pLoc(L) P)) ~> K) env(Env)
   = k(o(Xc, L, OEnv [Xc,Env]) ~> K) env(Env) .

  ***
  *** This takes an object and returns a reference to it, at the
  *** same time putting it into the store.
  ***
  eq k(o(X, L, OEnv) ~> returnObjectRef ~> K)  =
     k(allocRefFor(o(X, L, OEnv)) ~> K)  .

  *** Allocate storage for all the fields in an object. Note we need cases here for
  *** each basic type and for more complex types (inline object descriptors,
  *** repetitions, etc).
  *** TODO: We still need to add logic for booleans here, and any other basic types
  *** (like floats if we add them). We should also factor these out into separate modules,
  *** since this module is getting fairly large.
  op allocFields : AttributePart -> ContinuationItem .
  op setVirtual : -> ContinuationItem .
  eq k(allocFields(emptyAtP) ~> K) = k(K) .

*** Named patterns
  eq k(allocFields( (X : OD)) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> assignTo(X) ~> K) env(Env [X,L]) .
  eq k(allocFields( (X : OD) ; AtP) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> assignTo(X) ~> allocFields(AtP) ~> K)
     env(Env [X,L]) .
```

```
*** Virtual patterns, with a reference to the pattern name
  eq k(allocFields( (X :< Xc)) ~> K) env(Env) =
     k(Xc ~> setVirtual ~> assignTo(X) ~> K) env(Env) .
  eq k(allocFields( (X :< Xc) ; AtP) ~> K) env(Env) =
     k(Xc ~> setVirtual ~> assignTo(X) ~> allocFields(AtP) ~> K) env(Env) .

*** Virtual patterns, with an anonymous pattern supplied
  eq k(allocFields( (X :< OD)) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> setVirtual ~> assignTo(X) ~> K) env(Env [X,L]) .
  eq k(allocFields( (X :< OD) ; AtP) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> setVirtual ~> assignTo(X) ~> allocFields(AtP) ~> K)
     env(Env [X,L]) .

*** Extended virtual patterns, with a reference to the pattern name
  eq k(allocFields( (X ::< Xc)) ~> K) env(Env) =
     k(Xc ~> setVirtual ~> assignTo(X) ~> K) env(Env) .
  eq k(allocFields( (X ::< Xc) ; AtP) ~> K) env(Env) =
     k(Xc ~> setVirtual ~> assignTo(X) ~> allocFields(AtP) ~> K) env(Env) .

*** Extended virtual patterns, with an anonymous pattern supplied
  eq k(allocFields( (X ::< OD)) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> setVirtual ~> assignTo(X) ~> K) env(Env [X,L]) .
  eq k(allocFields( (X ::< OD) ; AtP) ~> K) env(Env [X,L]) =
     k(canonObj(X, OD, (Env [X,L]), L) ~> setVirtual ~> assignTo(X) ~> allocFields(AtP) ~> K)
     env(Env [X,L]) .

*** General declarations (like @integer, or ^BankAccount)
  eq k(allocFields( (X : Dec)) ~> K) env(Env) =
     k(createDec(Dec) ~> assignTo(X) ~> K) env(Env) .
  eq k(allocFields( ((X : Dec) ; AtP)) ~> K) env(Env) =
     k(createDec(Dec) ~> assignTo(X) ~> allocFields(AtP) ~> K) env(Env) .

*** Operations for specific declaration types. This should be extended for other
*** types of values, such as booleans and reals.
  eq k(createDec(^ T) ~> K) env(Env [T,L]) =
     k(href(T, L, nothing) ~> K) env(Env [T,L]) .

  eq k(createDec(@ integer) ~> K) = k(int(0) ~> K) .

*** Flip the virtual flag on a pattern to on
  eq k(pt(virtual(B:Bool) P) ~> setVirtual ~> K) =
     k(pt(virtual(true) P) ~> K) .

***
*** For this case, we need to create the object and invoke the do code,
*** but we also need to return the object itself, since this is a
*** stack-allocation. So, we need to "hop" the object around the
*** invocation, since a typical invocation would just throw the
*** object reference away and return whatever was in the exit block.
***
```

```
  op invokeWReturn : Value -> ContinuationItem .
  op createPatDec : -> ContinuationItem .
  eq k(createDec(@ Xc) ~> K) env(Env) =
     k(Xc ~> createPatDec ~> Env ~> K) env(Env) .
  eq k(pt(P) ~> createPatDec ~> K) =
     k(pt(P) ~> createObject(pt(P)) ~> finalizeObject(pt(P)) ~> invokeWReturn(pt(P)) ~> K) .
  eq k(O ~> invokeWReturn(pt(P)) ~> Env ~> K) =
     k(O ~> returnObjectRef ~> invokable(pt(P)) ~> Env ~> O ~> K) .

  *** An anonymous pattern definition, provided inline. Create an anonymous
  *** name, canonicalize it (see code in CLASS-SEMANTICS for more info), and
  *** then return it, which will put the pattern on top of the declaration.
  *** Then we just process as above, for where just a pattern name is given.
  var N : Nat .
  eq t(k(createDec(@ OD) ~> K) env(Env) TS) nextAnon(N) =
     t(k(bindToNothingWLoc(anonObj(N)) ~> canonHolder(anonObj(N),OD,Env) ~> anonObj(N) ~>
         createPatDec ~> Env ~> K) env(Env) TS)
       nextAnon(N + 1) .

  *** below are for repetitions
  eq k(createDec([ E ]^ T)  ~> K) = k(E ~> createRepetition(^ T)  ~> K) .
  eq k(createDec([ E ]@ T)  ~> K) = k(E ~> createRepetition(@ T)  ~> K) .
  eq k(createDec([ E ]@ OD) ~> K) = k(E ~> createRepetition(@ OD) ~> K) .

endm

***
*** Semantics to allow pattern variables to be defined
*** and used in correct expression positions (such as
*** assignments, invocations, etc). Note that, since patterns
*** are just values in our semantics, we don't need to go to
*** this much trouble, but wrapping pattern var values in a
*** pvar allows us to ensure things are used in more logical
*** ways, and would also allow to to dynamically (if needed)
*** check to ensure that the pattern being assigned is a child
*** of the declared pattern var type (this checking is not
*** currently done).
***
mod PATTERN-VARIABLE-SEMANTICS is
  including PATTERN-MEMBERSHIP-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including NEW-SEMANTICS .

  var E : Exp . var K : Continuation . var X : Name .
  var Env : Env . vars L PL : Location . var T : Type .
  var P : Pattern . var V : Value .

***
*** A pattern holder -- used for variables defined
*** like a : ##b. This way we can keep track of the
*** defined type of a, while also keeping track of the
*** current assigned pattern. The triple includes:
*** type name x location of type definition x pattern value
***
 op pvar : Type Location Value -> Value .

*** Pattern variable declarations
  eq k(createDec(## T) ~> K) env(Env [T,L]) =
     k(pvar(T, L, nothing) ~> K) env(Env [T,L]) .

*** Retrieve pattern def from a pattern var
  eq k(pvar(T,L,V) ~> getPattern ~> K) = k(V ~> K) .

*** Handle assigning to a pattern variable
  op assignToPVar : Name -> ContinuationItem .
  eq k((E -> (X ##)) ~> K) = k((E,X) ~> assignToPVar(X) ~> nothing ~> K) .
  eq k((pt(P), pvar(T,PL,V)) ~> assignToPVar(X) ~> K) =
     k(pvar(T,PL,pt(P)) ~> assignTo(X) ~> K) .

*** Handle invoking from a pattern variable
  eq k(pvar(T,PL,pt(P)) ~> lookupForInvoke ~> K) =
     k(pt(P) ~> lookupForInvoke ~> K) .
endm

***
*** Convert other data types into string form. Do it in a module like this so we don't
*** introduce strings above where we don't need them.
***
fmod TO-STRING is
  protecting CONVERSION .
  protecting INT .
  protecting STRING .

  op toString : Int -> String .

  var N : Int . var S : String .

  eq toString(N) = string(N,10) .
endfm

***
*** Handle output. Not sure if we will have any input yet (the Beta book doesn't
*** even cover the output, it just happens to be in the examples and very useful).
***
mod IO-SEMANTICS is
  protecting GENERAL-IMP-SYNTAX .
  protecting GENERIC-EXP-SEMANTICS .
  protecting IO-SYNTAX .
  protecting TO-STRING .
  protecting BOOL-IMP-SEMANTICS .
  op writeexp : -> ContinuationItem .
  ops puttextC putintC putbooleanC : -> ContinuationItem .

  op strval : String -> Value .

  var Str : String . var E : Exp .
  var K : Continuation . var SL : StringList .
  var I : Int . var X : Name . var IL : IntList .
  var TS : PLThreadState .

  eq k((E -> puttext) ~> K) = k(E ~> puttextC ~> K) .
  rl t(k(strval(Str) ~> puttextC ~> K) TS) output(SL) => t(k(K) TS) output(SL :: Str) .
  eq k(str(Str) ~> K) = k(strval(Str) ~> K) .

  rl t(k(newline ~> K) TS) output(SL) => t(k(K) TS) output(SL :: "\n") .

  eq k((E -> putint) ~> K) = k(E ~> putintC ~> K) .
  rl t(k(int(I) ~> putintC ~> K) TS) output(SL) => t(k(K) TS) output(SL :: toString(I)) .

  eq k((E -> putboolean) ~> K) = k(E ~> putbooleanC ~> K) .
  rl t(k(bool(true) ~> putbooleanC ~> K) TS) output(SL) => t(k(K) TS) output(SL :: "true") .
  rl t(k(bool(false) ~> putbooleanC ~> K) TS) output(SL) => t(k(K) TS) output(SL :: "false") .
endm

fmod INT-SET is
  including INT .
  sort IntSet .
  subsort Int < IntSet .
  op empty : -> IntSet .
  op _#_ : IntSet IntSet -> IntSet [assoc comm id: empty] .
  op _in_ : Int IntSet -> Bool .
  var I : Int .  var Is : IntSet .
  eq I in I # Is = true .
  eq I in Is = false [owise] .
  eq I # I = I .
endfm

fmod COUNTER-SET is
  including INT-SET .
  sorts Counter CounterSet .
  subsort Counter < CounterSet .
  op empty : -> CounterSet .
  op [_,_] : Int Int -> Counter .
  op __ : CounterSet CounterSet -> CounterSet [assoc comm id: empty] .
  op _-_ : IntSet CounterSet -> IntSet .
  var I : Int .  var Is : IntSet .  var N : Nat .  var Cs : CounterSet .
  eq (I # Is) - ([I,N] Cs) = Is - Cs .
  eq Is - (empty).CounterSet = Is .
endfm
```

```
mod ALTERNATION-SEMANTICS is
  including CONCURRENT-OBJECT-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including COUNTER-SET .
  including NEW-SEMANTICS .

  vars P P' P'' : Pattern . vars E E' : Exp . vars K K' : Continuation .
  var O : Object . vars L L' PL : Location .
  var T : Type . vars X X' : Name .
  vars V V' : Value .
  vars Env Env' : Env .
  var Xc : Name .
  var Vl : ValueList .  var Cs : CounterSet .
  var Is : IntSet .  var N : Nat .  var Nz : NzNat .  var I : Int .
  var CI : ContinuationItem . vars TS TS' TS'' : PLThreadState .
  var ImpL : ImpList .
  var OD : ObjDesc . var Ns : NameList .

***
*** A dref is like an href, but is for dynamic references to component
*** objects instead of item objects. This just helps us to tell them apart.
*** The equations below are just duplicates of the href definitions.
***
  op dref : Type Location Value -> Value .
  eq k((oref(L),dref(T,PL,V)) ~> assignToRef(X) ~> K) =
     k(dref(T,PL,oref(L)) ~> assignTo(X) ~> K) .
  eq k(dref(T,L,oref(L')) ~> lookup(X) ~> K) =
     k(ldir(L) ~> block(dref(T,L,oref(L'))) ~> lookup(X) ~> K) .
  eq k(pt(pname(Xc) P) ~> block(dref(T,L,oref(L'))) ~> lookup(X) ~> K) =
     k(deref(oref(L')) ~> dispatch(T,L,oref(L'),pt(pname(Xc) P)) ~> lookup(X) ~> K) .

***
*** When given a pattern name, get back the pattern. With an inline
*** declaration, first canonicalize it and give it a name, then treat
*** it as if we looked it up. For references, just create the reference
*** holder. Don't evaluate it here yet.
***
*** TODO: Do we need to recover the environment below? Need to think about
*** that.
***
  op toFreeze : -> ContinuationItem .
  eq k(createDec(@ | Xc) ~> K) =
     k(Xc ~> toFreeze ~> K) .
  eq t(k(createDec(@ | OD) ~> K) env(Env) TS) nextAnon(N) =
     t(k(bindToNothingWLoc(anonObj(N)) ~> canonHolder(anonObj(N),OD,Env) ~>
         anonObj(N) ~> toFreeze ~> K) env(Env) TS) nextAnon(N + 1) .
  eq k(createDec(^ | T) ~> K) env(Env [T,L]) =
     k(dref(T, L, nothing) ~> K) env(Env [T,L]) .
```

```
***
*** Create a frozen version of the object -- hold on to the pattern,
*** but don't actually evaluate it yet. We won't evaluate it until
*** we attach it to the execution stack (see altern).
***
*** TODO: Do we need to save the environment here? Or can we just
*** use the environment at the "unfreeze" time?
***
  op frozenObj : Value Env -> Value .
  op unfreeze : -> ContinuationItem .

  eq k(pt(P) ~> toFreeze ~> K) env(Env) =
     k(frozenObj(pt(P),Env) ~> K) env(Env) .

***
*** We have the special syntax altern(Component) to differentiate
*** a standard pattern reference/insertion from an altern call.
*** When we see altern(X), we want to grab the pattern for X and
*** attach it to the current execution stack (see Chapter 13
*** of the Beta book).
***

***
*** Keep track of the active alternating object, so we can
*** assign the stack back to the object location. Don't use
*** names, in case we have scoping issues.
***
  op foraltern : Name -> ContinuationItem .
  eq k(altern(X) ~> source(Vl) ~> K) = k(X ~> foraltern(X) ~> source(Vl) ~> K) .
  eq k(altern(X) ~> K) = k(altern(X) ~> source(nil) ~> K) .

***
*** If we are unfreezing an object definition for alternation, we need to
*** create a copy of the object and save the current execution environment.
***
  op alternState : PLThreadState -> PLThreadStateAttribute .
  op activeAltern : Name -> PLThreadStateAttribute .
  op alternPattern : Value -> PLThreadStateAttribute .
  op unfreezeState : -> Continuation .
  op frozenState : PLThreadState -> Value .

***
*** The following looks fairly complex, but is actually pretty straight-forward.
*** If we encounter a frozen object on top of a foraltern, that means we are
*** trying to start the object. To do so, we need to create it based on its
*** pattern and save the current state into alternState so, when we either
*** suspend this alternation or finish, we can get the current state back.
*** Note that we store the pattern definition of the current alternation
*** pattern as well, since if we finish we want to "reset" to the start of
*** the do block (for generators, for instance -- see Ch 13.2)
```

```
***
  eq t(k(frozenObj(pt(P),Env) ~> foraltern(X) ~> source(Vl) ~> K) env(Env') alternState(TS')
       activeAltern(X') alternPattern(V) TS)  =
     t(k(pt(P) ~> createObject(pt(P)) ~> finalizeObject(pt(P)) ~> returnObjectRef
       ~> invokable(pt(P)) ~> source(Vl) ~> unfreezeState) env(Env)
       alternState(k(K) env(Env') alternState(TS') activeAltern(X')
       alternPattern(V) TS) activeAltern(X) alternPattern(pt(P)) TS) .

***
*** Similar to the above, but we have already created the object and at some
*** point suspended it. So, we just grab the state back out and start running
*** from where we left off. One thing to note is we always go back "up"
*** first to the controlling object, so we don't risk losing information
*** when we write a new alternState attribute (we don't need the old state
*** saved there anymore, and in fact don't even save it below).
***
  eq t(k(frozenState(k(K') env(Env) alternPattern(pt(P entryList(Ns))) TS) ~>
     foraltern(X) ~> source(nil) ~> K) env(Env') alternState(TS') activeAltern(X') TS'')
   = t(k(K') env(Env) alternPattern(pt(P entryList(Ns))) alternState(k(K) env(Env')
     alternState(TS') activeAltern(X') TS'') activeAltern(X) TS) .

  eq t(k(frozenState(k(K') env(Env) alternPattern(pt(P entryList(Ns))) TS) ~>
       foraltern(X) ~> source(Vl) ~> K) env(Env') alternState(TS') activeAltern(X') TS'')
   = t(k(Vl ~> assignTo Ns ~> K') env(Env) alternPattern(pt(P entryList(Ns))) alternState(k(K)
       env(Env') alternState(TS') activeAltern(X') TS'') activeAltern(X) TS) [owise] .

***
*** Two variations: one where we save and recover the stack, one where we just
*** recover the stack.
***
  op recoverStack : ValueList -> Continuation .
  op saveAndRecoverStack : ValueList -> ContinuationItem .

***
*** When we hit the bottom of the alternating process's execution stack,
*** we should return any values (if we have them) and recover the prior
*** stack. We need to save something, since we don't want to pick up
*** again from the last save point, but we are mainly interested in
*** just picking up where we left off in the caller. Note: in Mjolner
*** if you try to rerun a complete alternating process, you crash.
***
  eq k(Vl ~> unfreezeState) = k(recoverStack(Vl)) .
  eq k(unfreezeState) = k(recoverStack(nil)) .

***
*** When we suspend an alternating process, we need to save off all the
*** state so we can grab it later. Also, we need to evaluate all the
*** names in the exit list so we can return their values
***
  eq k(suspend ~> K) alternPattern(pt(exitList(Ns) P)) =
     k(Ns ~> saveAndRecoverStack(nil) ~> K) alternPattern(pt(exitList(Ns) P)) .
  eq k(Vl ~> saveAndRecoverStack(nil) ~> K) =
     k(saveAndRecoverStack(Vl) ~> K) .

  eq t(k(recoverStack(Vl)) env(Env) alternState(k(K') env(Env') activeAltern(X')
       alternState(TS'') TS') activeAltern(X) TS)
   = t(k((frozenState(k(recoverStack(Vl)) env(Env) TS)) ~> assignTo X ~> Vl ~> K') env(Env')
       activeAltern(X') alternState(TS'') TS') .

  eq t(k(saveAndRecoverStack(Vl) ~> K) env(Env) alternState(k(K') env(Env')
       activeAltern(X') alternState(TS'') TS') activeAltern(X) TS)
   = t(k((frozenState(k(K) env(Env) TS)) ~> assignTo X ~> Vl ~> K') env(Env')
       activeAltern(X') alternState(TS'') TS') .

endm

mod THREAD-SEMANTICS is
  including ALTERNATION-SEMANTICS .
  including CONCURRENT-OBJECT-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including COUNTER-SET .
  including NEW-SEMANTICS .
  op holds : CounterSet -> PLThreadStateAttribute .
  op busy : IntSet -> PLStateAttribute .
  op die : -> Continuation .

  var P : Pattern . vars E E' : Exp . vars K K' : Continuation .
  var O : Object . var L : Location .
  var T : Type . vars X X' : Name .
  var Mem : Store .

  vars Env Env' : Env .
  var Xc : Name .
  var Vl : ValueList .  var Cs : CounterSet .
  var Is : IntSet .  var N : Nat .  var Nz : NzNat .  var I : Int .
  var CI : ContinuationItem . vars TS TS' TS'' : PLThreadState .

***
*** This is almost like a macro to expand createForInvoke
*** into all the steps needed to do so
***
  op createForInvoke : -> ContinuationItem .
  eq k(pt(P) ~> createForInvoke ~> K) =
     k(pt(P) ~> createObject(pt(P)) ~> finalizeObject(pt(P)) ~>
       returnObjectRef ~> invokable(pt(P)) ~> K) .

***
*** .fork creates a new thread of execution in which we run a given
*** component (a special pattern specified with the | syntax). We run this
*** in the current environment and object context. When created, the
*** component was "frozen" so it would not execute immediately -- we
*** unfreeze it here.
***

  op tofork : -> ContinuationItem .
  eq k((E .fork) ~> K) = k(E ~> tofork ~> K) .

***
*** When we unfreeze a frozen object, actually create it based
*** on the creation pattern (essentially, we just delay object
*** creation until we unfreeze it here).
***
***
*** TODO: (Baris) Maybe we should continue using Env'
  eq t(k(frozenObj(pt(P),Env) ~> tofork ~> K) env(Env') holds(Cs) innerList(Vl) TS) =
     t(k(K) env(Env') holds(Cs) innerList(Vl) TS)
     t(k(pt(P) ~> createForInvoke ~> die) env(Env) holds(empty) innerList(nil) TS) .

***
*** If we find a value on top of the die continuation, or just
*** a naked die continuation, remove any locks and remove
*** the die-ing thread.
*** TODO: Is removing all the acquired locks specified in the book?
  var PLS : PLStateAttribute .
  eq t(k(Vl ~> die) TS) PLS = PLS .
  eq t(k(die) TS) PLS = PLS .

  op sem : Int -> Value .
  ops semP semV : -> ContinuationItem .
  op waitingFor : Location -> ContinuationItem .
  op allowContextSwitch : -> ContinuationItem .
  eq k(createDec(@ Semaphore) ~> K) = k(sem(0) ~> K) .
  eq k((X .P) ~> K) = k(semaphorRead(X) ~> semP ~> K) .
  eq k((X .V) ~> K) = k(semaphorRead(X) ~> semV ~> K) .

  *** Covering different cases for P (aka decrease, aka acquire):
  ***   1) semaphor value is greater than zero
  ***   2) semaphor value is zero
  ***   3) semaphor value is less than zero, and semaphor is held by current thread
  ***   4) semaphor value is less than zero, and semaphor is NOT held by current thread
  ***      which means the thread has to wait for the value to become zero.
 ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds(Cs) TS) mem(Mem)
   = t(k(allowContextSwitch ~> K) holds(Cs) TS) mem(Mem[loc(N) <- sem(I + (-1))])
   if I > 0 .
  eq t(k((loc(loc(N)), sem(0)) ~> semP ~> K) holds(Cs) TS) mem(Mem)
   = t(k(allowContextSwitch ~> K) holds([N, 0] Cs) TS) mem(Mem[loc(N) <- sem(-1)]) .
 ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds([N, 0] Cs) TS) mem(Mem)
   = t(k(allowContextSwitch ~> K) holds([N, 0] Cs) TS) mem(Mem[loc(N) <- sem(I + (-1))])
   if I < 0 .
 ceq t(k((loc(loc(N)), sem(I)) ~> semP ~> K) holds(Cs) TS) mem(Mem)
```

```
   = t(k(waitingFor(loc(N)) ~> semP ~> K) holds(Cs) TS) mem(Mem)
   if I < 0 [owise] .

  *** Covering different cases for V (aka increase, aka release):
  ***   1) semaphor value is less than -1, and semaphor is held by current thread
  ***   2) semaphor value is -1, which means that the semaphor will be released and
  ***      a waiting thread will be notified.
  ***   3) semaphor value is greater than -1
 ceq t(k((loc(loc(N)), sem(I)) ~> semV ~> K) holds([N, 0] Cs) TS) mem(Mem)
   = t(k(allowContextSwitch ~> K) holds([N, 0] Cs) TS) mem(Mem[loc(N) <- sem(I + 1)])
   if I < -1 .
 ceq t(k((loc(loc(N)), sem(I)) ~> semV ~> K) holds(Cs) TS) mem(Mem)
   = t(k(allowContextSwitch ~> K) holds(Cs) TS) mem(Mem[loc(N) <- sem(I + 1)])
   if I > -1 .
  eq t(k((loc(loc(N)), sem(-1)) ~> semV ~> K) holds([N, 0] Cs) TS) mem(Mem)
     t(k(waitingFor(loc(N)) ~> K') TS')
   = t(k(allowContextSwitch ~> K) holds(Cs) TS) mem(Mem[loc(N) <- sem(0)])
     t(k((loc(loc(N)), sem(0)) ~> K') TS') .
  eq t(k((loc(loc(N)), sem(-1)) ~> semV ~> K) holds([N, 0] Cs) TS) mem(Mem) t(k(K') TS')
   = t(k(allowContextSwitch ~> K) holds(Cs) TS) mem(Mem[loc(N) <- sem(0)]) t(k(K') TS') [owise] .

  rl k(allowContextSwitch ~> K) => k(K) .
endm

mod BETA-SEMANTICS is
  including BETA-SYNTAX .
  including GENERIC-EXP-SEMANTICS .
  including ARITH-IMP-SEMANTICS .
  including BOOL-IMP-SEMANTICS .
  including FOR-SEMANTICS .
  including IF-SEMANTICS .
  including SEQ-COMP-SEMANTICS .
  including LOOKUP-SEMANTICS .
  including ASSIGNMENT-SEMANTICS .
  including REPETITION-SEMANTICS .
  including PATTERN-MEMBERSHIP-SEMANTICS .
  including PATTERN-VARIABLE-SEMANTICS .
  including METHOD-SEMANTICS .
  including CLASS-SEMANTICS .
  including NEW-SEMANTICS .
  including IO-SEMANTICS .
  including ALTERNATION-SEMANTICS .
  including THREAD-SEMANTICS .
  including LABELED-IMP-SEMANTICS .

  op eval : Exp -> [String] .
  op eval : ObjDesc -> [String] .
  op eval : ObjDesc StringList -> [String] .
  op eval : PatternDecl -> [String] .
  op eval : PatternDecl StringList -> [String] .
  op eval* : Exp -> [String] .
  op eval* : ObjDesc -> [String] .
  op eval* : ObjDesc StringList -> [String] .
  op eval* : PatternDecl -> [String] .
  op eval* : PatternDecl StringList -> [String] .

  op [_] : PLState -> [String] .
  op *[_]* : PLState -> [String] .
  op objdef : -> Value .

  var E : Exp .  var V : Value .  var S : PLState .
  var OD : ObjDesc . var Vl : ValueList .
  var X : Name . var IL : IntList . vars SL SL' : StringList .
  var TS : PLThreadState . vars N N' : Nat . var Mem : Store .

  op startState : -> PLState .
  op startThreadState : -> PLThreadState .
  eq objdef = pt(pname(Object) entryList(emptyNL) exitList(emptyNL) doCode(inner)
             pLoc(loc(0)) parent(HierTop) attributes(emptyAtP) pEnv([Object,loc(0)])
             parentLoc(loc(0)) virtual(false)) .
  eq startState = nextLoc(2) mem([loc(0),objdef][loc(1),nothing]) nextAnon(0)
                 output(nilS) busy(empty) .
  eq startThreadState = env([Object,loc(0)])
                 obj(o(Object,loc(0),[Object,[Object,loc(0)]['mainProg,loc(1)]['main,loc(2)]]))
                 class(objdef) innerList(nil) holds(empty) alternState(empty)
                 activeAltern('mainProg) alternPattern(nothing) .

  eq eval(E) = [t(k(E ~> stop) startThreadState) startState ] .
  eq eval(OD) = eval(OD,nilS) .
  eq eval(OD,SL) = eval( ('main : OD),SL) .
  eq eval((X : OD)) = eval((X : OD),nilS) .
  eq eval((X : OD),SL) = [t(k((X : OD) ~> (& X) ~> stop) startThreadState ) input(SL) startState ] .
  eq [t(k(Vl ~> stop) TS) S output(SL)] = concat(SL) .
  eq [t(k(stop) TS) S output(SL)] = concat(SL) . *** change to return output instead

  eq eval*(E) = *[t(k(E ~> stop) startThreadState) startState ]* .
  eq eval*(OD) = eval*(OD,nilS) .
  eq eval*(OD,SL) = eval*( ('main : OD),SL) .
  eq eval*((X : OD)) = eval*((X : OD),nilS) .
  eq eval*((X : OD),SL) = *[t(k((X : OD) ~> (& X) ~> stop) startThreadState)
     input(SL) startState ]* .
  eq *[t(k(Vl ~> stop) TS) busy(empty) output(SL) nextLoc(N) mem(Mem)
     input(SL') nextAnon(N')]* = concat(SL) .
  eq *[t(k(stop) TS) busy(empty) output(SL) nextLoc(N) mem(Mem) input(SL')
     nextAnon(N')]* = concat(SL) . *** change to return output instead

endm
```

C Model Checker Definition

```
in beta-semantics .
in model-checker

mod BETA-MODELCHECK is
  inc MODEL-CHECKER .
  inc BETA-SEMANTICS .
  subsort PLState < State .
  op deadlocked : -> Prop .
  op buildState : ObjDesc -> PLState .
  op filterThreads : PLState -> PLState .
  op filterThreads : PLState -> PLState .
  op inDeadlock : PLState -> [Bool] .

  vars PLS PLS' : PLState . var OD : ObjDesc . vars TS TS' : PLThreadState .
  var L : Location . var K : Continuation .
  eq t(k(waitingFor(L) ~> K) TS) PLS |= deadlocked = inDeadlock(filterThreads(PLS)) .
  eq filterThreads(t(k(stop) TS) PLS) = filterThreads(PLS) .
  eq filterThreads(t(k(waitingFor(L) ~> K) TS) PLS) = filterThreads(PLS) .
  eq filterThreads(PLS) = PLS [owise] .
  eq inDeadlock(t(TS) PLS) = false .
  eq inDeadlock(PLS) = true [owise] .

  eq buildState(OD) = t(k(('Main : OD) ~> (& 'Main) ~> stop) startThreadState )
     input(nilS) startState .
  ops initial1 initial2 initial3 initial4 : -> PLState .
  eq initial1 = buildState( (# p : (# do (str("In p") -> puttext) #) ;
                               q : (# do (str("In q") -> puttext) #) ;
                               a : @ | p ; b : @ | q ;
                               do (a .fork ; b .fork ) #)) .

  eq initial2 = buildState((# 's1 : @ Semaphore ;
             's2 : @ Semaphore ;
             p : (# do ('s1 .P ; 's2 .P ; str("In p") -> puttext ; 's2 .V ; 's1 .V) #) ;
             q : (# do ('s2 .P ; 's1 .P ; str("In q") -> puttext ; 's1 .V ; 's2 .V) #) ;
             a : @ | p ;
             b : @ | q ;
             do (a .fork ; b .fork )
            #)) .
  eq initial3 = buildState((# 's1 : @ Semaphore ;
             's2 : @ Semaphore ;
             p : (# do ('s1 .P ; 's2 .P ; str("In p") -> puttext ; 's2 .V ; 's1 .V) #) ;
             q : (# do ('s1 .P ; 's2 .P ; str("In q") -> puttext ; 's2 .V ; 's1 .V) #) ;
             a : @ | p ;
             b : @ | q ;
             do (a .fork ; b .fork )
            #)) .
endm
```

```
eof

red modelCheck(initial1, [] ~ deadlocked) .
***> should be true
red modelCheck(initial3, [] ~ deadlocked) .
***> should be true
red modelCheck(initial2, [] ~ deadlocked) .
***> should be false
```

D Sample Programs

```
***
*** Lookup of value from remote name
***
rew eval(
(# a : @ (# b : @ integer ; do (5 -> b) #) ;
   do (a . b -> putint)
#)) .

rew eval(
(# a : @ (#
     b : @ (#
       c : @ (#
         d : @ ( # e : @ integer ; do (5 -> e) #) ;
       #) ;
     #) ;
  #) ;
   do (a . b . c . d . e -> putint)
#)) .

***
*** Write value to remote name
***
rew eval(
(# a : @ (# b : @ integer ; do (5 -> b) #) ;
   do (10 -> a . b -> putint)
#)) .

rew eval(
(# a : @ (#
     b : @ (#
       c : @ (#
         d : @ ( # e : @ integer ; do (5 -> e) #) ;
       #) ;
     #) ;
  #) ;
   do (20 -> a . b . c . d . e ; a . b . c . d . e  -> putint)
#)) .
```

```
rew eval(
(# a : @ (#
     b : @ (#
       c : @ (#
         d : @ ( # e : @ integer ; do (5 -> e) #) ;
       #) ;
     #) ;
  #) ;
   do (20 -> a . b . c . d . e -> putint)
#)) .

rew eval(
(# a : @ (#
     b : @ (#
       c : @ (#
         d : @ ( # e : @ integer ;
                   f : @ integer ;
                   g : @ integer ;
                   do (5 -> e ; 6 -> f ; 7 -> g) #) ;
       #) ;
     #) ;
  #) ;
   do (
     str("Initial values") -> puttext ;
     a . b . c . d . e -> putint ;
     str(" ") -> puttext ;
     a . b . c . d . f -> putint ;
     str(" ") -> puttext ;
     a . b . c . d . g -> putint ;
     str("Ready to assign...") -> puttext ;
     20 -> a . b . c . d . e ;
     50 -> a . b . c . d . f ;
     2521 -> a . b . c . d . g ;
     str("Values after assignment") -> puttext ;
     a . b . c . d . e -> putint ;
     str(" ") -> puttext ;
     a . b . c . d . f -> putint ;
     str(" ") -> puttext ;
     a . b . c . d . g -> putint )
#)) .

***
*** In/out parameters for alternation (cannot use
*** parameters on .fork calls, at least not in
*** Mjolner, and no examples from Beta book have them)
***
rew eval(
(# i : @ integer ;
   b : @ | (# do (('L : i + 5 -> i ; suspend ; restart 'L : 'L))
              exit i #) ;
   do (10 -> i ; altern(b) -> i ; i -> putint)
#)).

rew eval(
(# i : @ integer ;
   b : @ | (# do (('L : i + 5 -> i ; suspend ; restart 'L : 'L))
              exit i #) ;
   do (10 -> i ; altern(b) -> i ; i -> putint ; altern(b) -> i ;
       str(" ") -> puttext ; i -> putint )
#)).

rew eval(
(# i : @ integer ;
   b : @ | (# n : @ integer ;
              enter n
              do (('L : n + 5 -> n ; suspend ; restart 'L : 'L))
              exit n #) ;
   do (10 -> i ; i -> altern(b) -> i ; i -> putint )
#)).

rew eval(
(# i : @ integer ;
   b : @ | (# n : @ integer ;
              enter n
              do (('L : n + 5 -> n ; suspend ; restart 'L : 'L))
              exit n #) ;
   do (10 -> i ; i -> altern(b) -> i ; i -> putint ; i -> altern(b) -> i ;
       str(" ") -> puttext ; i -> putint )
#)).

rew eval(
(# i : @ integer ;
   b : @ | (# n : @ integer ;
              enter n
              do (('L : n + 5 -> n ; suspend ; restart 'L : 'L))
              exit n #) ;
   do (10 -> i ; i -> altern(b) -> i ; i -> putint ; i -> altern(b) -> i ;
       str(" ") -> puttext ; i -> putint ; altern(b) -> i ; str(" ") ->
       puttext ; i -> putint  )
#)).

rew eval((#
'ForTo : (# 'First, 'Last, 'Index : @ integer ;
             enter 'First, 'Last
             do ('First -> 'Index ;
                 (('Loop : inner ; 'Index + 1 -> 'Index ;
                    (if ('Index gt 'Last)
                     // True then (leave 'Loop)
                     else (restart 'Loop) if) : 'Loop))) #) ;
do ((1,100) -> ins('ForTo (# enter 'First, 'Last do ('Index -> putint ; str(" ") -> puttext) #)))
```

```
#)) .

rew eval((#
'Cycle : (# do (('Loop : inner ; restart 'Loop  : 'Loop)) #) ;
'ForTo : (# 'First, 'Last, 'Index : @ integer ;
             enter 'First, 'Last
             do ('First -> 'Index ;
                 (('Loop : inner ; 'Index + 1 -> 'Index ;
                    (if ('Index gt 'Last)
                     // True then (leave 'Loop)
                     else (restart 'Loop) if) : 'Loop))) #) ;
'Factorial : @ |
  (# 'T : [100] @ integer ; 'N, 'Top, 'Ret : @ integer ;
     enter 'N
     do ( 1 -> 'Top -> 'T[1] ;
       ins ('Cycle (#
         do ((if ('Top lt 'N)
              // True then ( ('Top + 1,'N) ->
                 ins('ForTo (# enter 'First, 'Last do 'T['Index - 1] * 'Index -> 'T['Index] #)) ;
                 'N -> 'Top) if) ;
            'N + 1 -> 'N ;
            'T['N - 1] -> 'Ret ;
            suspend) #))
        )
     exit 'Ret
  #) ;
'F : @ integer ;
do (4 -> altern('Factorial) -> 'F ; 'F -> putint ; str(" ") -> puttext ;
    altern('Factorial) -> 'F ; 'F -> putint ; str(" ") -> puttext ;
    3 -> altern('Factorial) -> 'F ; 'F -> putint ; str(" ") -> puttext)
#)) .

***
*** Semaphores
***
search eval*( (# 's1 : @ Semaphore ;
              p : (# do ('s1 .P ;
                         str("In p1 ") -> puttext ;
                         str("In p2 ") -> puttext ;
                         's1 .V)
                   #) ;
              q : (# do ('s1 .P ;
                         str("In q1 ") -> puttext ;
                         str("In q2 ") -> puttext ;
                         's1 .V)
                   #) ;
              a : @ | p ;
              b : @ | q ;
              do (a .fork ; b .fork )
            #) ) =>! ST:[String] .
```

```
***
*** Inserted objects
***
rew eval(
(# do ( 1 -> ins((# a : @ integer ;
                enter a
                do (a + 5 -> a)
                exit a
              #)) -> putint
      )
#)) .

***
*** Control pattern
***
rew eval(
(# 'Cycle : (# do (('Loop : inner ; restart 'Loop  : 'Loop)) #) ;
   x : @ integer ;
   do (1 -> x ;
       (('L : ins('Cycle (# do (if x
                               // 10 then (leave 'L)
                               else (x + 1 -> x)
                               if)
                          #)) : 'L)) ;
       x -> putint
   )
#)) .

***
*** Labels
***
rew eval(
(# a : @ integer ;
   do ( 1 -> a ;
        (('top :
          (if a
             // 10 then (str("Leaving...") -> puttext ; leave 'top)
             else (a + 1 -> a ; str("Restarting...") -> puttext ; restart 'top)
          if)
        : 'top)) ;
        str("Left...") -> puttext
      )
#)) .

***
*** "Tuple"-style assignments
***
rew eval(
(# a,b,c,d : @ integer ;
   'plusone : (# a : @ integer ;
                 enter a
                 do (a + 1 -> a)
                 exit a #) ;
   do ( 5 -> a ;
        6 -> b ;
        (a -> & 'plusone, b -> & 'plusone) -> (c,d) ;
        c -> putint ;
        str(" ") -> puttext ;
        d -> putint
      )
#)) .

***
*** Some more simple concurrency examples
***
rew eval*(
(# x : @ integer ;
   'plusone : (# do (x + 1 -> x) #) ;
   y, z : @ | 'plusone ;
   do (y .fork ; z .fork ; x -> putint)
#)) .

search [10] eval*(
(# x : @ integer ;
   'plusone : (# do (x + 1 -> x) #) ;
   y, z : @ | 'plusone ;
   do (y .fork ; z .fork ; x -> putint)
#)) =>! ST:[String] .

search [10] eval*(
(# x : @ integer ;
   'plusone : (# do (x + x -> x) #) ;
   y, z : @ | 'plusone ;
   do (1 -> x ; y .fork ; z .fork ; x -> putint)
#)) =>! ST:[String] .

***
*** Virtual patterns
***
rew eval(
(# x : (# a :< (# do (5 -> putint ; str(" ") -> puttext ; inner) #) ;
          b : (# do (21 -> putint ; str(" ") -> puttext ; inner) #) ;
        #) ;
   v : x (# a ::< (# do (6 -> putint) #) ;
            b : (# do (25 -> putint ; str(" ") -> puttext ; inner) #) ;
          #) ;
   y : ^ x ;
   do (& v [] -> y [] ; & y . a)
```

```
#)) .

rew eval(
(# x : (# a :< (# do (5 -> putint ; str(" ") -> puttext ; inner) #) ;
          b : (# do (21 -> putint ; str(" ") -> puttext ; inner) #) ;
        #) ;
   v : x (# a ::< (# do (6 -> putint) #) ;
            b : (# do (25 -> putint ; str(" ") -> puttext ; inner) #) ;
          #) ;
   y : ^ x ;
   do (& v [] -> y [] ; & y . b)
#)) .

rew eval(
(# x : (# a :< (# do (5 -> putint ; str(" ") -> puttext ; inner) #) ; #) ;
   v : x (# a ::< (# do (6 -> putint) #) ; #) ;
   y : ^ x ;
   z : @ x ;
   do (& v [] -> y [] ; & z . a ; str(" ") -> puttext ; & y . a)
#)) .

rew eval(
(# x : (# a :< (# do (5 -> putint ; str(" ") -> puttext ; inner) #) ; #) ;
   v : x (# a ::< (# do (6 -> putint) #) ; #) ;
   w : x (# a ::< (# do (7 -> putint) #) ; #) ;
   y : ^ x ;
   z : @ x ;
   do (& v [] -> y [] ; & y . a ; & w [] -> y [] ; str(" ") -> puttext ; & y . a)
#)) .

rew eval(
(# x : (# a :< (# do (5 -> putint ; str(" ") -> puttext ; inner) #) ; #) ;
   v : x (# a ::< (# do (6 -> putint) #) ; #) ;
   y : ^ x ;
   z : @ x ;
   do (& v [] -> y [] ; & z . a ; & y . a ; & x [] -> y [] ; & y . a)
#)) .

rew eval( ( # p : ( # a,b,c : @ integer ; # ) ; q,r,s : @ p ; # ) ) .

rew eval( ( # p : ( # a,b,c : @ integer ; # ) ; q : @ p ; # ) ) .

rew eval( ( # p : ( # a,b,c : @ integer ; # ) ; q : ^ p ; # ) ) .

rew eval( ( # p : ^ Object ; # ) ) .

rew eval( (# a,b,c : @ integer ; #) ) .

rew eval( ( # p : ( # a,b,c : @ integer ; d : ^ p ; # ) ;
              q : p ( # e : @ integer ; # ) ;
              z : @ q ; # )) .

rew eval( ( # a, b, c : @ integer ; do (5 -> a ; 6 -> b ; a -> putint)  # ) ) .

rew eval( ( # 'myPattern : @ (# a, b, c : @ integer ;
                                d : @ (# do (5 -> a ;
                                             6 -> b
                                            )
                                       #) ;
                              #) ;
               do ('myPattern . a -> putint)
            # ) ) .

rew eval( ( # a, b, c : @ integer ;
              do (5 -> a ;
                  str("The value of a is:") -> puttext ;
                  newline ;
                  a -> putint)  # ) ) .

rew eval( ( # p : ( # a,b,c : @ integer ; d : ^ p ; # ) ;
              q : p ( # e : @ integer ;
                        do ( 5 -> a ; 6 -> e ; (a + e) -> putint) # ) ;
              z : @ q ;
              do (& q) # )) .

rew eval( (# p : (# a : @ integer ; do (str("In p") -> puttext ) #) ;
             q : ^ p ;
             do ( str("Ready to create") -> puttext ;
                  & p [] -> q [] ;
                  str("Done") -> puttext ) #) ) .

rew eval( ( # p : ( # a,b,c : @ integer ;
                      do (str("Assigning defaults") -> puttext ;
                      5 -> a ;
                      6 -> b ;
                      7 -> c) # ) ;
              q : @ p ;
              do ((q . b) -> putint ) # ) ) .

rew eval( ( # p : ( # a,b,c : @ integer ;
                      d : (# do ((c +  1) -> c) #) ;
                      do (str("Assigning defaults") -> puttext ; 5 -> a ; 6 -> b ; 7 -> c) # ) ;
              q : @ p ;
              do ((q . c) -> putint ; & (q . d) ) # ) ) .

rew eval( ( # p : ( # a,b,c : @ integer ;
                      d : (# do ((c +  1) -> c) #) ;
                      do (str("Assigning defaults") -> puttext ;
                        5 -> a ; 6 -> b ; 7 -> c) # ) ;
              q : @ p ;
              do ((q . c) -> putint ; (& (q . d)) ; (q . c) -> putint ) # ) ) .

rew eval( ( # a : [3]@ integer ; do (str("a[1] : ") -> puttext ; a[1] -> putint ;
     str(", a.range : ") -> puttext ; a .range -> putint ;
     8 -> a .new ;
     str(", a.range : ") -> puttext ; a .range -> putint ;
     2 -> a[1] ;
     str(", a[1] : ") -> puttext ; a[1] -> putint ;
     4 -> a .extend ;
     str(", a.range : ") -> puttext ; a .range -> putint ;
     9 -> a[12] ;
     str(", a[a.range] : ") -> puttext ; a[a .range] -> putint
     )  # ) ) .

rew eval( ( # a : [100]@ integer ; 'sum : @ integer ;
                                 do ( (for i : a .range repeat (i -> a[i]) for) ;
                                      0 -> 'sum ;
      (for i : a .range repeat ('sum + a[i] -> 'sum) for) ;
             str("sum : ") -> puttext ; 'sum -> putint
    ) # ) ) .

rew eval( ( # a : [6]@ integer ; do ((8,3,4) -> a[4 $ 6] ;
     str("  a[1] : ") -> puttext ; a[1] -> putint ;
     str(", a[2] : ") -> puttext ; a[2] -> putint ;
     str(", a[3] : ") -> puttext ; a[3] -> putint ;
     str(", a[4] : ") -> puttext ; a[4] -> putint ;
     str(", a[5] : ") -> puttext ; a[5] -> putint ;
     str(", a[6] : ") -> puttext ; a[6] -> putint ;
                                     (a[3 $ 5]) -> a[1 $ 3] ;
     str(" | a[1] : ") -> puttext ; a[1] -> putint ;
     str(", a[2] : ") -> puttext ; a[2] -> putint ;
     str(", a[3] : ") -> puttext ; a[3] -> putint ;
     str(", a[4] : ") -> puttext ; a[4] -> putint ;
     str(", a[5] : ") -> puttext ; a[5] -> putint ;
     str(", a[6] : ") -> puttext ; a[6] -> putint

     )  # ) ) .

rew eval( (# a,b,c : @ integer ;
             do (
          (1,2,3) -> (a,b,c) ;
  str("The value of a is: ") -> puttext ; a -> putint ; str(" ") -> puttext ;
  str("The value of b is: ") -> puttext ; b -> putint ; str(" ") -> puttext ;
  str("The value of c is: ") -> puttext ; c -> putint ; str(" ") -> puttext
)
           #)) .

rew eval( (# 'sum2 : (# a,b,c : @ integer ;
                        enter a,b
 do ( a + b -> c ; str("The sum is: ") -> puttext ; c -> putint ; str(" ") -> puttext )
```

```
      #) ;
             'r1, 'r2 : @ integer ;
     do (
          (1,4) -> & 'sum2 ;
  (10,20) -> & 'sum2
)
          #) ) .

rew eval( (# 'sum2 : (# a,b,c : @ integer ;
                         enter a,b
 do ( a + b -> c ; str("The sum is: ") -> puttext ; c -> putint ; str(" ") -> puttext )
 exit c
      #) ;
             'r1, 'r2 : @ integer ;
     do (
          (1,4) -> & 'sum2 -> 'r1 ;
  (10,20) -> & 'sum2 -> 'r2 ;
  str("The value of r1 is: ") -> puttext ; 'r1 -> putint ; str(" ") -> puttext ;
  str("The value of r2 is: ") -> puttext ; 'r2 -> putint ; str(" ") -> puttext
)
          #) ) .

rew eval( (# 'sum2 : (# a,b,c : @ integer ;
                         enter a,b
 do ( a + b -> c ; str("The sum is: ") -> puttext ; c -> putint ; str(" ") -> puttext )
 exit c
      #) ;
             'r1, 'r2, 'r3 : @ integer ;
     do (
          (1,4) -> & 'sum2 -> 'r1 ;
  (10,20) -> & 'sum2 -> 'r2 ;
  ( (100,45) -> & 'sum2 , (30,50) -> & 'sum2 ) -> & 'sum2 -> 'r3 ;
  str("The value of r1 is: ") -> puttext ; 'r1 -> putint ; str(" ") -> puttext ;
  str("The value of r2 is: ") -> puttext ; 'r2 -> putint ; str(" ") -> puttext ;
  str("The value of r3 is: ") -> puttext ; 'r3 -> putint ; str(" ") -> puttext
)
          #) ) .

rew eval( (# 'pass_it_on : (# a : @ integer ;
                              enter a
      do (str("In pass it on: ") -> puttext ;
                                  a -> putint ;
                                  str(", changing to: ") -> puttext ;
                                  a + 1 -> a ;
                                  a -> putint ;
                                  str(", next will be: ") -> puttext ;
                                  a + 1 -> putint ;
                                  str(" ") -> puttext
                                 )
      exit a
```

```
    #) ;
      a : @ integer ;
     do ( 1 -> & 'pass_it_on -> & 'pass_it_on -> & 'pass_it_on ->
                          & 'pass_it_on -> & 'pass_it_on -> a ;
          str("The final value is: ") -> puttext ; a -> putint
 )
   #)) .

rew eval( (# 'pass_it_on : (# a : @ integer ;
                              enter a
      do (str("In pass it on: ") -> puttext ; a + 1 -> a ; a -> putint ; str(" ") -> puttext)
      exit a
    #) ;
      a : @ integer ;
     do ( 1 -> & 'pass_it_on ;
          1 -> & 'pass_it_on ;
  1 -> & 'pass_it_on -> & 'pass_it_on -> & 'pass_it_on -> a ;
          str("The final value is: ") -> puttext ; a -> putint
 )
   #)) .

rew eval( (# f : (# a : @ integer ;
                   do (5 -> a -> putint) exit a  #) ;
             b : @ integer ;
             do (& f -> b -> putint) #)) .

rew eval( (# f : (# a : @ integer ; enter a do (a -> putint) #) ; do (3 -> & f) #)) .

rew eval( (# a : (# do (str("In a before ") -> puttext ; inner ;
                        str("In a after ") -> puttext) #) ;
             b : a (# do (str("In b before ") -> puttext ; inner ;
                          str("In b after 1 ") -> puttext ;
                  inner ; str("In b after 2 ") -> puttext) #) ;
     c : b (# do (str("In c before ") -> puttext ; inner ;
                          str("In c after ") -> puttext) #) ;
     do (& c)
   #)) .

rew eval( (# a : (# do (str("In a before ") -> puttext ; inner ;
                        str("In a after ") -> puttext) exit a #) ;
             b : a (# do (str("In b before ") -> puttext ; inner ;
                          str("In b after 1 ") -> puttext ;
                  inner ; str("In b after 2 ") -> puttext) exit b #) ;
     c : b (# do (str("In c before ") -> puttext ; inner ;
                          str("In c after ") -> puttext) exit c #) ;
     do ((if ((& c))
          // (a) then (str("          type a") -> puttext)
          // (b) then (str("          type b") -> puttext)
          // (c) then (str("          type c") -> puttext)
  else (str("       no type") -> puttext)
if))
   #)) .

***
*** Basic pattern membership tests
***
rew eval( (# a : (# do (inner) #) ;
             b : @ a ;
     do ((if (b ##)
          // (a ##) then (str("type a") -> puttext)
  else str("type not a") -> puttext
          if))
   #)) .

rew eval( (# a : (# do (inner) #) ;
             c : (# do (inner) #) ;
             d : (# do (inner) #) ;
             b : @ c ;
     do ((if (b ##)
          // (a ##) then (str("type a") -> puttext)
                  // (c ##) then (str("type c") -> puttext)
                  // (d ##) then (str("type d") -> puttext)
  else str("type not a, c, or d") -> puttext
          if))
   #)) .

rew eval( (# a : (# do (inner) #) ;
             c : (# do (inner) #) ;
             d : (# do (inner) #) ;
             e : (# do (inner) #) ;
             b : @ e ;
     do ((if (b ##)
          // (a ##) then (str("type a") -> puttext)
                  // (c ##) then (str("type c") -> puttext)
                  // (d ##) then (str("type d") -> puttext)
  else str("type not a, c, or d") -> puttext
          if))
   #)) .

***
*** Pattern assignment tests
***
rew eval( (# a : (# do (str("In a") -> puttext) #) ;
             b : a (# do (str("In b") -> puttext) #) ;
             c : ## a ;
             do (a ## -> c ## ; & c )
#)) .

rew eval( (# a : (# do (str("In a") -> puttext) #) ;
             b : a (# do (str("In b") -> puttext) #) ;
             c : ## a ;
             do (a ## -> c ## ; & c )
#)) .

***
*** SAMPLE FROM CHAPTER 2
***
rew eval(
(# 'Account :
    (# 'balance : @ integer ;

       'Deposit :
         (# 'amount : @ integer ;
            enter 'amount
            do 'balance + 'amount -> 'balance
            exit 'balance
         #) ;

       'Withdraw :
         (# 'amount : @ integer ;
            enter 'amount
            do 'balance - 'amount -> 'balance
            exit 'balance
         #) ;
    #) ;

  'newBalance : @ integer ;
  'account1 : @ 'Account ;

  do ( str("Prior balance: ") -> puttext ;
       'account1 . 'balance -> putint ;
       500 -> & 'account1 . 'Deposit -> 'newBalance ;
       str(", New balance: ") -> puttext ;
       'newBalance -> putint
     )
#)
) .

rew eval(
(# x : (# a : (# do (5 -> putint) #) ; #) ;
   y : ^ x ;
   do (& x [] -> y [] ; & y . a)
#)) .

rew eval(
(# x : (# a : (# do (5 -> putint ; inner) #) ; #) ;
   y : ^ x ;
   z : @ x ;
   do (& x [] -> y [] ; & z . a ; & y . a)
#)) .
```

```
rew eval(
(# x : (# a :< (# do (5 -> putint ; inner) #) ; #) ;
   v : x (# a ::< (# do (6 -> putint) #) ; #) ;
   y : ^ x ;
   z : @ x ;
   do (& x [] -> y [] ; & z . a ; & y . a)
#)) .

***
*** Deterministic alternation
***
rew eval( (# p : (# do (str("In p") -> puttext ; suspend ; str("In p again") -> puttext ) #) ;
             q : (# do (str("In q") -> puttext) #) ;
             a : @ | p ;
             b : @ | q ;
             do (altern(a) ; altern(b) ; altern(a) ) #) ) .

rew eval( (# p : (# do (str("In p") -> puttext ; str("In p again") -> puttext ) #) ;
             q : (# do (str("In q") -> puttext) #) ;
             a : @ | p ;
             b : @ | q ;
             do (altern(a) ; altern(b) ; altern(a) ) #) ) .

rew eval*( (# p : (# do (str("In p") -> puttext ; suspend ; str("In p again") -> puttext ) #) ;
              q : (# do (str("In q") -> puttext) #) ;
              a : @ | p ;
              b : @ | q ;
              do (altern(a) ; altern(b) ; altern(a) ) #) ) .

***
*** Concurrent processes
***
rew eval*( (# p : (# do (str("In p") -> puttext) #) ;
              q : (# do (str("In q") -> puttext) #) ;
              a : @ | p ;
              b : @ | q ;
              do (a .fork ; b .fork ) #) ) .

search eval*( (# p : (# do (str("In p") -> puttext) #) ;
                 q : (# do (str("In q") -> puttext) #) ;
                 a : @ | p ;
                 b : @ | q ;
                 do (a .fork ; b .fork ) #) ) =>! ST:[String] .

search[10] eval*((#
  a,c : @ integer ;
  't1 : (# do (1 -> a) #) ;
  t : @ | 't1 ;
  do (t .fork ;
      (('L : (if a
                // 0 then (c + 1 -> c ; restart 'L)
                else (leave 'L) if) : 'L)) ;
      c -> putint
     )
#)) =>! ST:[String] .

search[1] eval*((#
  a,c : @ integer ;
  't1 : (# do (1 -> a) #) ;
  t : @ | 't1 ;
  do (t .fork ;
      (('L : (if a
                // 0 then (c + 1 -> c ; restart 'L)
                else (leave 'L) if) : 'L)) ;
      c -> putint
     )
#)) =>! ST::String such that ST::String == "100" .


***
*** More complex example with virtual/non-virtual patterns, inline patterns,
*** inner calls, etc.
***
rew eval(
(# i : ^ a ;
***   j : @ c ;
   q : (# do(str(" <Q> ") -> puttext ; inner ; str(" </Q> ") -> puttext) #) ;
   t : q (# do(str(" <T> ") -> puttext ; inner ; str(" </T> ") -> puttext) #) ;
   a : (# p : q (# do(str(" <a.p> ") -> puttext ; inner ; str(" </a.p> ") ->  puttext) #) ;
          v :< q ;
  ***v : (# do(str(" <a.v> ") -> puttext ; inner ; str(" </a.v> ") -> puttext) #) ;
  w : (# do(& v) #) ;
          do(str(" <a> ") -> puttext ; inner ; str(" </a> ") -> puttext)
       #) ;
   c : a (# p : q (# do( str(" <c.p> ") -> puttext ; inner ; str(" <c.p> ") -> puttext ) #) ;
            v ::< t (# do( str(" <c.v> ") -> puttext ; inner ; str(" <c.v> ") -> puttext ) #) ;
            ***v ::< t ;
          do(str("<c>") -> puttext ; str("</c>") -> puttext)
       #) ;
   do(
      & c [] -> i [] ;
      & i . p ; *** non-virtual
      str("  |  ") -> puttext ;
      & i . v ; *** virtual
      str("  |  ") -> puttext ;
      & i . w
     )
#)
) .
```