WADT'08, LNCS 5486, pp 187-205. 2009

Towards a Module System for K*

Mark Hills and Grigore Roşu

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

Abstract. Research on the semantics of programming languages has yielded a wide array of notations and methodologies for defining languages and language features. An important feature many of these notations and methodologies lack is *modularity*: the ability to define a language feature once, insulating it from unrelated changes in other parts of the language, and allowing it to be reused in other language definitions. This paper introduces ongoing work on modularity features in K, an algebraic, rewriting logic based formalism for defining language semantics.

Key words: language semantics, rewriting logic, modularity, K

1 Introduction

One important aspect of formalisms for defining the semantics of programming languages is modularity. Modularity is generally expressed as the ability to add new language features, or modify existing features, without having to modify unrelated semantic rules. For instance, when designing a simple expression language, one may want to use structural operational semantics (SOS) [31] to define the semantics of addition:

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \text{(EXP-PLUS-L)}$$

$$\frac{e_2 \rightarrow e_2'}{n_1 + e_2 \rightarrow n_1 + e_2'} \qquad \text{(EXP-PLUS-R)}$$

$$n_1 + n_2 \rightarrow n, \text{where } n = n_1 + n_2 \qquad \text{(EXP-PLUS)}$$

Further extending the language, one may want to add variables. The standard way to do this is to define a store, mapping names to values, with rules for

* Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

WADT'08, LNCS 5486, pp 187-205. 2009

binding values to names (not shown here) and a rule to retrieve the current value of a binding:

$$\langle x, \sigma \rangle \rightarrow \langle n, \sigma \rangle, \text{where } n = \sigma(x) \qquad \text{(VAR-LOOKUP)}$$

With this change to the language, even though the rules for plus do not actually reference the store, since it is now part of the SOS configuration the existing rules must be modified. As an example, rule `EXP-PLUS-L` becomes[1]:

$$\frac{\langle e_1, \sigma \rangle \rightarrow \langle e_1', \sigma \rangle}{\langle e_1 + e_2, \sigma \rangle \rightarrow \langle e_1' + e_2, \sigma \rangle} \qquad \text{(EXP-PLUS-L)}$$

Similar types of changes to existing rules need to be made to accommodate other unrelated language features, such as exceptions or function returns. Alternatively, similar changes may need to be made to add addition expressions to a different language with a different configuration, even if the different elements of the configuration are not used in the rules for addition.

All these changes are required because SOS is not modular. Improved support for modularity eliminates the need to make these changes, offering several advantages:

- Modular definitions of language features allow other parts of a language to change more easily, allowing existing feature definitions to remain unchanged in the face of unrelated modifications or additions;
- A modular definition of a language feature can be more easily reused in the definition of a different language which may be structured much differently;
- Modular definitions are easier to understand, since the rules given for a language construct only need to include the information needed by the rule, instead of including extraneous information used in other parts of the language (such as the store in the rules for plus).

For these reasons, improving modularity of language definitions has been a focus of research across multiple semantic formalisms. One example is modular structural operational semantics (MSOS) [27, 28], which solves the problem shown above by leveraging the labels on rule transitions, not normally used in SOS definitions of programming languages, to encode configuration elements, with the ability to elide unused parts of the configuration. This, and other work on modularity, will be discussed further in Section 4.

With a tool supported semantics, modularity can also be expressed as the ability to package language features into discreet reusable units, which can then be assembled when defining a language. This form of modularity depends on the first: it should be possible to plug the same feature into multiple definitions, even in cases where (unused) parts of the configuration are different. Additionally, it

[1] A more general version of this rule would use σ on the left and σ' on the right; here, by using σ on both left and right, we state that expressions do not alter the store, i.e. they do not have side effects.

WADT'08, LNCS 5486, pp 187-205. 2009

should be possible to provide clean interfaces to language features and to different parts of the configuration, something not required in monolithic definitions, or even in modular definitions written on paper.

This paper provides a high-level overview of ongoing work on adding modularity features to K [32], an algebraic, rewriting logic based formalism for programming language semantics. This work is focused on both aspects of modularity mentioned above, allowing the packaging of language features for reuse while insulating existing features from unrelated changes to the language definition.

The remainder of this paper is organized as follows. First, we provide a brief overview of term rewriting, equational logic, rewriting logic, and especially K in Section 2. Next, Section 3 introduces the module system through fragments of a simple imperative language. Section 4 then reviews related work, including the aforementioned MSOS, while in Section 5 we conclude and discuss future work.

2 Rewriting Logic

This section provides a brief introduction to term rewriting, rewriting logic, rewriting logic semantics, and K. Term rewriting is a standard computational model supported by many systems; rewriting logic [21, 20] organizes term rewriting modulo equations as a complete logic and serves as a foundation for programming language semantics using rewriting logic semantics [23, 24]. K [32] is a rewrite-based method for formally defining computation, here used to provide formal definitions for programming languages.

2.1 Term Rewriting

Term rewriting is a method of computation that works by progressively changing (rewriting) a term. This rewriting process is defined by a number of rules – potentially containing variables – which are each of the form: $l \to r$. A rule can apply to the entire term being rewritten or to a subterm of the term. First, a match within the current term is found. This is done by finding a substitution, θ, from variables to terms such that the left-hand side of the rule, l, matches part or all of the current term when the variables in l are replaced according to the substitution. The matched subterm is then replaced by the result of applying the substitution to the right-hand side of the rule, r. Thus, the part of the current term matching $\theta(l)$ is replaced by $\theta(r)$. The rewriting process continues as long as it is possible to find a subterm, rule, and substitution such that $\theta(l)$ matches the subterm. When no matching subterms are found, the rewriting process terminates, with the final term being the result of the computation. Rewriting, like other methods of computation, can continue forever.

There exist a plethora of term rewriting engines, including ASF+SDF [36], Elan [2], Maude [5], OBJ [10], and others. Rewriting is also a fundamental part of existing languages and theorem provers.

WADT'08, LNCS 5486, pp 187-205. 2009

2.2 Rewriting Logic

Rewriting logic [21, 20] is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the λ calculus with equivalence classes based on α and β equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency, where different orders of evaluation could lead to non-equivalent results, such as in the case of data races. The distinction between rules and equations is crucial for analysis, since terms which are equal according to equational deduction can all be collapsed into the same analysis state. Rewriting logic is connected to term rewriting in that the equations and rules of rewriting logic, of the form $l = r$ and $l \Rightarrow r$, respectively, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming both into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and a term and "executing" it.

2.3 Rewriting Logic Semantics

Rewriting logic semantics (RLS) [23, 24] builds upon the observation that programming languages can be defined as rewriting logic theories. By doing so, one gets essentially "for free" not only an interpreter and an initial model semantics for the defined language, but also a series of formal analysis tools obtained as instances of existing tools for rewriting logic. The work discussed in this paper has grown out of a subset of RLS referred to in the cited work as Continuation-Based Semantics, which treats computations as first-class entities in the semantics. This allows for the natural modeling of complex control flow constructs, such as exceptions and `call/cc`.

2.4 K

K [32], a general notation and technique for defining computation, is based on insights developed in the rewriting logic semantics project [23, 24], with some concepts inspired by abstract state machines (ASMs) [11], the chemical abstract machine (CHAM) [9], and continuations [34]. K provides some domain-specific abstractions and assumptions, exploited in this paper, to ease the definition of programming languages.

The idea underlying language semantics in K is to represent the program configuration as a *computational structure*. This structure contains the context needed for the computation, with elements of the context represented as multisets or lists each stored inside a K *cell*. Contexts can also be hierarchical, with one cell containing others. The context generally includes standard items found in configurations in other formalisms, such as environments, stores, etc, as well as items specific to the given semantics, including such items as analysis results for

WADT'08, LNCS 5486, pp 187-205. 2009

a semantics focused on program analysis. One regularly used cell, referred to as k, represents the current computation as a $\curvearrowright$-separated list of computational tasks, such as $t_1 \curvearrowright t_2 \curvearrowright ... \curvearrowright t_n$. Another, $\top$, represents the entire computational structure. In the rest of the paper, the computational structure will be referred to as just a computation.

A K definition consists of two types of sentences: structural equations and rewrite rules. Structural equations carry no computational meaning; instead, borrowing a concept from CHAMs, structural equations can *heat* and *cool* computations. When a computation is heated, it breaks into smaller pieces, exposing subexpressions of more complex expressions for evaluation. Cooling reverses this process, reassembling the (potentially modified) pieces into a computation with the same "shape". The following are examples of structural equations:

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$
$$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightleftharpoons b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$$

Note that, unlike in evaluation contexts, $\square$ is not a "hole," but rather part of a *KLabel*, carrying the obvious "plug" intuition; e.g., the *KLabels* involving $\square$ above are $\square + \_$ (in the first equation) and $\mathsf{if}\ \square\ \mathsf{then\_else\_}$ (in the second).

Many structural equations can be automatically generated by annotating constructs in the language syntax with *strict* attributes: a *strict* construct generates the appropriate equations for each strict argument. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute; for example, the two equations directly above correspond to the attributes *strict* for $\_ + \_$ (i.e., strict in all arguments, with the heating/cooling equations for the second operand not shown) and *strict*(1) for $\mathsf{if\_then\_else\_}$.

Rewrite rules represent actual steps of computation. The following are examples of rewrite rules:

$$i_1 + i_2 \rightarrow i, \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2$$
$$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$$
$$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$$

$$(\!|\underset{\cdot}{\underline{X\ :=\ V}}\rangle\!\rangle_k\ \langle\!\langle(X, L)\rangle\!\rangle_{env}\ \langle\!\langle(L, \underset{V}{\underline{\_}})\rangle\!\rangle_{mem}$$

Structural equations can be applied back and forth; for example, the first equation for $\_ + \_$ can be applied left-to-right to "schedule" a_1 for processing; once evaluated to i_1, the equation is applied in reverse to "plug" the result back in context, then a_2 is similarly scheduled, then its result i_2 plugged back into context, and then finally the rewrite rule can apply, representing an irreversible computational step. Special care must be taken so that side effects are propagated appropriately, by providing enough context to ensure that they happen only at certain designated times – generally when the side effecting operation could be the next computational step (could be, since with concurrency there could be multiple next computational steps).

WADT'08, LNCS 5486, pp 187-205. 2009

An example with side effects is shown in the last rewrite rule, where a value V is to be assigned to name X. The round bracket at the left, $(\!|$, represents the head of the list, forcing this rule to apply on when it will be the next step of this computation. The "pointed" bracket at the right, $|\!\rangle$, represents the rest of the list, i.e. the remainder of the computation (intuitively, it is pointed as a reminder that the list keeps going in that direction). Multisets are bracketed with $\langle\!|$ and $|\!\rangle$, indicating they conceptually continue in either direction. This is used here for both env and mem: env is a multiset of Name × Location pairs, while mem is a multiset of Location × Value pairs. Other notation includes $\_$, which represents an unnamed value (like in many functional languages), and $\cdot$, representing the identity (here, the list identity). Finally, changes can be represented by showing the entire changed term on the right of the $\rightarrow$, but can also be represented in a two-dimensional form, with the original term on top and any changes shown by underlining the changed part of the term and noting the change below. This rule then states: when X := V is the next computational step in this computation, if X is at location L in the environment, change the value at location L in the store to V (while ignoring the current value), and then "dissolve" the current computation, leaving the next item in k as the next computational step.

To ensure that rules are modular, it should be possible to continue using a rule, unchanged, when parts of the context not mentioned in the rule are modified or replaced. Given a specific rule, the easiest case to deal with is when the subterm matched by a rule remains the same but the surrounding context changes (for instance, by adding a new top-level cell). This case is handled naturally by term rewrite systems, since it is possible to match a subterm of the term, leaving the rest unnamed in the rule. This handles many common cases, including the motivating example given in Section 1. However, this does not handle changes to the hierarchical organization of the context. For instance, adding threads to a language requires having multiple k cells, representing the computation occurring in each thread, but only one store, leading to a revised rule for assignment like the following:

$$\langle\!| (\!| \underset{\cdot}{\underline{X\ :=\ V}} |\!\rangle_k \ \langle\!| (X, L) |\!\rangle_{env} |\!\rangle_t \ \langle\!| (L, \underset{V}{\underline{\_}}) |\!\rangle_{mem}$$

Beyond this, the configurations used in different languages will generally be quite different, and may be very complex. As an example, the configuration used in the KOOL language [13] is shown in Figure 1. Reusing this rule in KOOL would require more changes, here to move k into the control configuration item. To allow rules to be reused, both as a language evolves and in other languages, K uses *context transformers*. Using context transformers, only those portions of the configuration actually used in a rule need to be mentioned. For instance, the assignment rule shown originally can remain as it was, without explicitly needing to show that the computation and environment are both in the same thread (or, in the case of KOOL, that the computation is inside the control), while the store is shared by the threads. The transformers then transform the rule so that it will match the configuration hierarchy assembled for the language, with the

WADT'08, LNCS 5486, pp 187-205. 2009

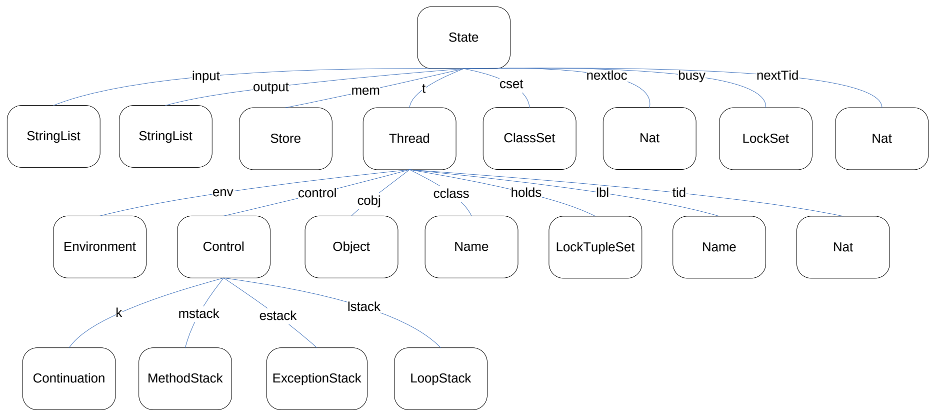

Fig. 1. KOOL State Infrastructure

intervening configuration items added automatically by the transformers, based on the defined configuration for the language.

3 The K Module System

While context transformers focus on making individual K rules modular, they do nothing to address the practical challenge of packaging up rules into reusable units. This is the purpose of the K module system.

```
module Path/Name
  imports Some/Mod, Other/Mod with { attribs } .
  exports sort SExp, op _op_ : SImp SImp' -> SImp' .
  requires sort SReq, op _^ : SReq -> SReq  .

  sort Loc .
  sortalias Store = Map(Loc,Value) .
  subsort SSub < SSup .

  var V : Value . var Store[0-9']* : Store .

  op _someop_ : SomeSort SomeSort -> SomeSort .
  eq [OptEqName] T = T'    [ where optional side-conditions ] .
  rl [OptRlName] T2 => T3  [ where optional side-conditions ] .
end module
```

Fig. 2. Generic Module Format

The module system in K is being designed to support a general module syntax incorporating the entire range of functionality needed when defining the

WADT'08, LNCS 5486, pp 187-205. 2009

semantics of a language, including the definition of abstract syntax, configuration items, the semantics of language features, and the final collection of features that make up a specific language. In theory, this would allow a single, monolithic module to include definitions of all aspects of the semantics. However, to provide for a better separation of these constructs into more granular modules, and to allow for construct-specific defaults and syntax, specialized module formats for various constructs are being defined, with a translation into the more general syntax. These module formats are illustrated with fragments of the definition of a simple imperative language, while the general module format is shown in Figure 2. A complete definition of this language is given in Appendices A and B; note that the definition given in this section differs slightly to better illustrate features of the module system.[2]

3.1 Semantic Entities

Semantic entities in K definitions include configuration items, such as environments and stores, and sorts or operations used during computations, such as computation items and values. A simple example is shown in Figure 3, which uses subsorting to allow K integers (modules starting with `K` are provided as part of K) to be treated as K values. By default, this declaration is available by any other module that imports `Int`.

```
module Int
  imports K/Value, K/Int .
  subsort Int < Value .
end module
```

Fig. 3. Semantic Entity: Integer Values

Another example is shown in Figure 4. This shows the definition of an environment, which provides a mapping from names to locations (a store then maps locations to values; the separation easily allows features like nested scopes and reference parameters for functions). Like in Figure 3, an existing K definition, in this case for sort `Name`, is imported. Instead of similarly importing a specific definition of locations (sort `Loc`), module `Env` uses `requires`, meaning that, when the language is finally assembled, one module must provide sort `Loc`. This allows the module to state a requirement without also stating the module that satisfies that requirement, allowing different modules to be used in different languages.

Since K provides lists, multisets, and maps by default, we can immediately refer to maps from sort `Name` to sort `Loc`; `sortalias` lets us give this sort a name, `Env`, which can then be used in the remainder of the definition. The `var`

[2] Note to reviewers: this sentence, and the referenced appendices, will be removed from the final version of the paper if accepted.

WADT'08, LNCS 5486, pp 187-205. 2009


```
module Env
  imports K/Name .
  requires sort Loc .
  sortalias Env = Map(Name,Loc) .
  var Env[0-9']* : Env .
end module
```

Fig. 4. Semantic Entity: Environments

declaration allows the definition of a variable pattern: `Env`, followed by 0 or more numbers or primes, will be used to represent entities of sort `Env` (e.g., `Env`, `Env8a`, `Env'`, etc.). Variables used in other modules that import `Env` will then be identified as being of sort `Env` when they match the variable pattern.[3].

3.2 Abstract Syntax

Before defining the semantics of language constructs, the abstract syntax of those constructs needs to be defined. This is done using abstract syntax modules, which are defined using a tag of `[Syntax]` after the module name. A first example of an abstract syntax module is the syntax for arithmetic expressions, shown in Figure 5. One way to define the sort of arithmetic expressions would be to define a new sort which could be made a subsort of `Exp`; here, instead, the sort `Exp`, imported from module `Exp`, is renamed to `AExp` using a sort renaming directive on the import of module `Exp`. A var pattern to refer to arithmetic expressions is then defined similarly to that for environments, shown above.

```
module Exp/AExp[Syntax]
  imports Exp[Syntax] with { sort Exp renamed AExp } .
  var AE[0-9'a-zA-Z]* : AExp .
end module
```

Fig. 5. Abstract Syntax: Arithmetic Expressions

A second abstract syntax module, defining the addition construct, is shown in Figure 6. Syntax is defined using mixfix notation with an algebraic notation similar to that used in Maude or SDF (although note that `op` is not required on syntax definitions). To increase modularity, it is recommended that each module define only one language construct, although it is possible to define multiple constructs in the same module.

[3] This capability is present in some systems, such as SDF, but not in others, such as Maude.

WADT'08, LNCS 5486, pp 187-205. 2009


```
module Exp/AExp/Plus[Syntax]
  imports Exp/AExp[Syntax] .
  _+_ : AExp AExp -> AExp .
end module
```

Fig. 6. Abstract Syntax: Plus

3.3 Semantic Rules

Once the syntax has been defined, the semantics of each construct need to be defined as well. One explicit goal of the module system is to allow different semantics to be easily defined for each language construct. For instance, it should be possible to define a standard dynamic/execution semantics, a static/typing semantics, and potentially other semantics manipulating different notions of value (for instance, various notions of abstract value used during analysis).

Figure 7 shows an example of a module defining the dynamic semantics of a language feature, here integer addition. Normally a semantics module will implicitly import the related syntax module. Here, since we are modifying the attributes on an imported operator, we need to explicitly import the syntax module. Two attributes are modified. First, we note that the operator is now strict in all arguments, which will automatically generate the structural heating and cooling equations. Second, we use extends to automatically "hook" the semantics of the feature to the builtin definition of integer addition. This completely defines integer addition in the language, so no rules are needed.

```
module Exp/AExp/Plus[Dynamic]
    imports Exp/AExp/Plus[Syntax]
            with { op _+_ now strict, extends + [Int * Int -> Int] } .
end module
```

Fig. 7. Dynamic Semantics: Plus

Figure 8 shows semantics for the same feature, but this time the static semantics (for type checking) are defined. Like in Figure 7, the operator for plus is changed to be strict. In this case, though, the values being manipulated are types, not integers, so we also need to import the types and use them in the two rules shown. Here, the first rule is for when an expression is type correct: the two operands are both integers, so the result of adding them is also an integer. If one of the operands is not an integer (checked in the side-condition), the rule will cause a type called fail, representing a type error, to propagate.[4]

[4] An alternative would be to issue an error message and return the expected type in the hope of finding additional errors

WADT'08, LNCS 5486, pp 187-205. 2009


```
module Exp/AExp/Plus[Static] is
  imports Exp/AExp/Plus[Syntax] with { op _+_ now strict } .
  imports Types .
  rl int + int => int .
  rl T + T' => fail [where T =/= int or T' =/= int] .
end module
```

Fig. 8. Static Semantics: Plus

Finally, Figure 9 shows the dynamic semantics of blocks. Here, no changes are made to the imported syntax, so there is no need to import the `Stmt/Block[Syntax]` module explicitly. In this language, blocks provide for nested scoping, so we want to ensure that the current environment is restored after the code inside the block executes. This is done by capturing the current environment, `Env`, and placing it on the computation in a `restoreEnv` computation item. The rule for `restoreEnv`, not shown here, will replace the current environment with its saved environment when it becomes the first item in the computation. The bracket and arrow notation shown here in the rule is the equivalent of $\langle\!|\, \frac{\texttt{begin}\ S\ \texttt{end}}{S \curvearrowright \texttt{restoreEnv(Env)}} \,|\!\rangle_k \ \langle\!|\texttt{Env}|\!\rangle_{env}$

```
module Stmt/Block[Dynamic] is
  imports Stmt[Syntax], K/K, Env .
  rl k(| [begin S end ==> S ~> restoreEnv(Env)] |>  env(| Env |) .
end module
```

Fig. 9. Dynamic Semantics: Block

3.4 Language Definitions

Once the semantic entities, abstract syntax, and language semantics have been defined, they can be assembled into a language module, tagged `Language`.[5] An example is shown in Figure 10. The line `config =` defines the language configuration as a multiset, with each K cell given a name (such as `store` or `env`) and the sort of information in the cell (such as `Store` or `Env`). Cells can be nested, to represent the hierarchies of information that can be formed. Next, the `[[_]]` operator initializes this configuration, given an initial computation (`K`) representing the program to run. Finally, all the modules that make up the semantics are imported. `type=Dynamic` is a directive that states that all imported modules in this `imports` are tagged with the `Dynamic` tag, and is equivalent to `imports Exp/AExp/Name[Dynamic], Exp/AExp/Plus[Dynamic]`, etc.

[5] At this point a language cannot import another language definition, but we expect this to change soon.

WADT'08, LNCS 5486, pp 187-205. 2009


```
module Imp[Language]
  imports K/Configuration, K/K, K/Location, K/Value,
          Env, Store, Int, Bool .
  config = top(store(Store) env(Env) k(K) nextLoc(Loc)) .

  op [[_]] : K -> Configuration .
  eq [[ K ]] = top(store(empty) env(empty) k(K) nextLoc(initLoc)) .

  imports type=Syntax Exp/AExp/Num, Exp/BExp/Bool .
  imports type=Dynamic Exp/AExp/Name, Exp/AExp/Plus,
    Exp/BExp/LessThanEq, Exp/BExp/Not, Exp/BExp/And,
    Stmt/Sequence, Stmt/Assign, Stmt/IfThenElse,
    Stmt/While, Stmt/Halt, Pgm .
end module
```

Fig. 10. Language Definition: IMP

4 Related Work

Modularity has long been a topic of interest in the language semantics community. Listed below are some of the more significant efforts, including comparisons with the work described in this paper where appropriate.

Action Semantics: One focus of Action Semantics [26] has been on creating modular definitions. The notation for writing action semantics definitions uses a module structure, while language features use *facets* to separate different language construct "concerns", such as updating the store or communicating between processes. A number of tools have been created for working with modular Action Semantics definitions, such as ASD [37], the Action Environment [35], the Maude Action Tool [6], an implementation using Montages [1], and Modular Monadic Action Semantics [39]. Other work has focused specifically on ensuring modules can be easily reused without change, both by using small, focused modules [7] (the approach taken in the K module system) and by creating a number of simpler reusable constructs generic to a large number of languages [29, 15].

ASMs: Montages [17] provides a modular way to define language constructs using Abstract State Machines (ASMs) [14, 33]. Each Montage (i.e., module) combines a graphical depiction of a language construct with information on the static and dynamic semantics of the feature. This has the advantage of keeping all information on a feature in one place, but limits extensibility, since it is not possible (as it is in K) to provide multiple types of dynamic or static semantics to the same feature without creating a new Montage.

Denotational Semantics: One effort to improve modularity in denotational semantics definitions has been the use of Monads [25]. This has been most evident in work on modular, semantics-based definitions of language interpreters and

WADT'08, LNCS 5486, pp 187-205. 2009

compilers, especially in the context of languages such as Haskell [38, 16, 19, 18] and Scheme [8]. Monads have also been used to improve the modularity of other semantic formalisms, such as Modular Monadic Action Semantics [39], which provided a monadic semantics in place of the original, non-modular SOS semantics underlying prior versions of Action Semantics [26].

MSOS: The focus of MSOS [27, 28] has been on keeping the benefits of SOS definitions while defining rules in a modular fashion. This is done by moving information stored in SOS configurations, such as stores, into the labels on transition rules, which traditionally have not been used in SOS definitions of languages. This, along with techniques that allow parts of the label to be elided if not used by a rule, allow the same rule to be used both when unrelated parts of the configuration change and when the rule is introduced into a language with a different configuration. A recent innovation, Implicitly-Modular SOS (I-MSOS) [30], allows more familiar SOS notation while still providing the benefits of MSOS.

Rewriting Logic Semantics: Beyond the work done on K, Maude has also been used as a platform to experiment with other styles of semantics, enabling the creation of modular language definitions. This includes work on action semantics, with the Maude Action Tool cited above, and MSOS, using the Maude MSOS Tool [4]. Work on defining Eden [12], a parallel variant of Haskell, has focused on modularity to allow for experimentation with the degree of parallelism and the scheduling algorithm used to select processes for execution. General work on modularity of rewriting logic semantics definitions [22, 3] has focused on defining modular features that need not change as a language is extended.

5 Conclusion and Future Work

In this paper we have presented ongoing work on modularity in K language definitions. This includes work on modularity features of individual K rules, ensuring they can be defined once and reused in a variety of contexts, and work on an actual module system for K, providing a technique to easily package and reuse individual language features while building a language.

One major, ongoing component of this work is developing tool support for the module system. Although small modules improve reuse, the large number of modules this leads to can make it challenging to work with language definitions, something noted in similar work on tool support for Action Semantics [35]. For K, work on tool support includes the ongoing development of an Eclipse plugin to provide a graphical environment for the creation and manipulation of K modules. This will initially include editor support, a graph view of module dependencies, and the ability to view both the language features used to define a language and the various semantics defined for a specific language feature. Longer-term goals include the graphical assembly of language configurations and links to an online database of reusable modules.

WADT'08, LNCS 5486, pp 187-205. 2009

References

1. M. Anlauff, S. Chakraborty, P. W. Kutter, A. Pierantonio, and L. Thiele. Generating an action notation environment from Montages descriptions. *International Journal on Software Tools for Technology Transfer*, 3(4):431–455, 2001.
2. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *Proceedings of WRLA'98*, volume 15 of *ENTCS*, 1998.
3. C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. In *Proceedings of WRLA'04*, volume 117 of *ENTCS*, pages 393–416. Elsevier, 2005.
4. F. Chalub and C. Braga. Maude MSOS Tool. In *Proceedings of WRLA'06*, volume 176 of *ENTCS*, pages 133–146. Elsevier, 2007.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
6. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic. In *Proceedings of AMAST'00*, volume 1816 of *LNCS*, pages 407–421. Springer, 2000.
7. K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.
8. D. A. Espinosa. *Semantic Lego*. PhD thesis, 1995.
9. Gérard Berry and Gérard Boudol. The Chemical Abstract Machine. In *Proceedings of POPL'90*, pages 81–94. ACM Press, 1990.
10. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*. Kluwer, 2000.
11. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and Its Strategies for Defining a Framework for Analyzing Eden Semantics. In *Proceedings of WRS'06*, volume 174 of *ENTCS*, pages 119–137. Elsevier, 2007.
13. M. Hills and G. Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of RTA'07*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007.
14. J. Huggins. ASM-Based Programming Language Definitions. http://www.eecs.umich.edu/gasm/proglang.html.
15. J. Iversen and P. D. Mosses. Constructive Action Semantics for Core ML. *IEE Proceedings - Software*, 152(2):79–98, 2005.
16. D. J. King and P. Wadler. Combining Monads. In *Proceedings of Functional Programming'92*, Workshops in Computing, pages 134–143. Springer, 1992.
17. P. W. Kutter and A. Pierantonio. Montages Specifications of Realistic Programming Languages. *J. UCS*, 3(5):416–442, 1997.
18. S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In *Proceedings of ESOP'96*, volume 1058 of *LNCS*, pages 219–234. Springer, 1996.
19. S. Liang, P. Hudak, and M. P. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of POPL'95*, pages 333–343. ACM Press, 1995.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

WADT'08, LNCS 5486, pp 187-205. 2009

22. J. Meseguer and C. Braga. Modular Rewriting Semantics of Programming Languages. In *Proceedings of AMAST'04*, volume 3116 of *LNCS*, pages 364–378. Springer, 2004.
23. J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, volume 3097 of *LNAI*, pages 1–44. Springer, 2004.
24. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. Also appeared in *SOS '05*, volume 156(1) of *ENTCS*, pages 27–56, 2006.
25. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.
26. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
27. P. D. Mosses. Foundations of Modular SOS. In *Proceedings of MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer, 1999.
28. P. D. Mosses. Pragmatics of Modular SOS. In *Proceedings of AMAST'02*, volume 2422 of *LNCS*, pages 21–40. Springer, 2002.
29. P. D. Mosses. A Constructive Approach to Language Definition. *Journal of Universal Computer Science*, 11(7):1117–1134, 2005.
30. P. D. Mosses and M. J. New. Implicit Propagation in Structural Operational Semantics. In *SOS 2008*, 2008. Final version to appear in ENTCS.
31. G. D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, July-December 2004.
32. G. Roşu. K: A Rewriting-Based Framework for Computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois at Urbana-Champaign, 2007. Previous versions published as technical reports UIUCDCS-R-2006-2802 in December 2006, UIUCDCS-R-2005-2672 in 2005. K was first introduced in the context of Maude in Fall 2003 as part of a programming language design course (technical report UIUCDCS-R-2003-2897).
33. R. Stärk, J. Schmid, and E. Börger. *Java^{TM} and the Java^{TM} Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
34. C. Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000.
35. M. van den Brand, J. Iversen, and P. D. Mosses. An Action Environment. *Science of Computer Programming*, 61(3):245–264, 2006.
36. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.
37. A. van Deursen and P. D. Mosses. ASD: The Action Semantic Description Tools. In *Proceedings of AMAST'96*, volume 1101 of *LNCS*, pages 579–582. Springer, 1996.
38. P. Wadler. The Essence of Functional Programming. In *Proceedings of POPL'92*, pages 1–14. ACM Press, 1992.
39. K. Wansbrough and J. Hamer. A Modular Monadic Action Semantics. In *Proceedings of DSL'97*. USENIX, 1997.

WADT'08, LNCS 5486, pp 187-205. 2009

The following appendices show the definition of IMP, which is similar to the language used in this paper, although some language features defined in Section 3 were defined so as to illustrate certain features of the module system. Appendix A shows the K definition of IMP using a more mathematical notation, while Appendix B shows the modular K definition of IMP using ASCII notation. Note: these appendices will be removed from the final version of the paper if accepted.

A The K Definition of IMP

Figure 11 shows the K definition of the IMP language, which has been used as the running example for the presentation of the module system in this paper. This definition is discussed more fully in a technical report on K [32].

K-Annotated Syntax of IMP

$$
\begin{array}{rcll}
Int & ::= & \ldots \text{ all integer numbers} & \\
Bool & ::= & \mathsf{true} \mid \mathsf{false} & \\
Name & ::= & \text{all identifiers; to be used as names of variables} & \\
Val & ::= & Int & \\
AExp & ::= & Val \mid Name & \\
 & \mid & AExp + AExp & [strict,\ extends\ +_{Int \times Int \rightarrow Int}] \\
BExp & ::= & Bool & \\
 & \mid & AExp \leq AExp & [seqstrict,\ extends\ \leq_{Int \times Int \rightarrow Bool}] \\
 & \mid & \mathsf{not}\ BExp & [strict,\ extends\ \neg_{Bool \rightarrow Bool}] \\
 & \mid & BExp\ \mathsf{and}\ BExp & [strict(1)] \\
Stmt & ::= & Stmt; Stmt & [s_1; s_2 = s_1 \curvearrowright s_2] \\
 & \mid & Name := AExp & [strict(2)] \\
 & \mid & \mathsf{if}\ BExp\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt & [strict(1)] \\
 & \mid & \mathsf{while}\ BExp\ \mathsf{do}\ Stmt & \\
 & \mid & \mathsf{halt}\ AExp & [strict] \\
Pgm & ::= & Stmt; AExp &
\end{array}
$$

K Configuration and Semantics of IMP

$$
\begin{array}{rcl}
KResult & ::= & Val \\
K & ::= & KResult \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K] \\
Config & ::= & (\!| K |\!)_k \mid (\!| State |\!)_{state} \\
 & \mid & Val \mid [\![K]\!] \mid (\!| \mathsf{Set}[Config] |\!)_\top
\end{array}
$$

$$[\![p]\!] = (\!| (\!| p |\!)_k\ (\!| \emptyset |\!)_{state} |\!)_\top$$

$$\langle\!| (\!| v |\!)_k |\!\rangle_\top = v$$

$$(\!| \frac{x}{\sigma[x]} \rangle\!)_k\ (\!| \sigma |\!)_{state}$$

$$\mathsf{true}\ \mathsf{and}\ b \rightarrow b$$

$$\mathsf{false}\ \mathsf{and}\ b \rightarrow \mathsf{false}$$

$$(\!| \frac{x := v}{\cdot} \rangle\!)_k\ (\!| \frac{\sigma}{\sigma[v/x]} |\!)_{state}$$

$$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$$

$$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$$

$$(\!| \mathsf{while}\ b\ \mathsf{do}\ s \rangle\!)_k = (\!| \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot \rangle\!)_k$$

$$(\!| \mathsf{halt}\ i \rangle\!)_k = (\!| i |\!)_k$$

Fig. 11. K definition of IMP.

WADT'08, LNCS 5486, pp 187-205. 2009

B The Modular K Definition of IMP

A number of modules make up the definition of the IMP language. The first modules shown below make up semantic entities used in the definition.

```
module Env
  requires sort Name, sort Loc .
  sortalias Env = Map(Name,Loc) .
  var Env[0-9']* : Env .
end module

module Store
  requires sort Loc, sort Value .
  sortalias Store = Map(Loc,Value) .
  var Store[0-9']* : Store .
end module

module Int
  imports K/Int .
  requires sort Value .
  subsort Int < Value .
end module

module Bool
  sort Bool .
  ops true false : -> Bool .
end module
```

The next modules define the abstract syntax for IMP. This includes sorts for arithmetic expressions, boolean expressions, statements, and programs, as well as a number of syntactic entities (i.e., productions). Note that modules can import other modules of the same "type" (`Syntax`, `Dynamic`, etc) without needing to specify the type. If this would lead to an ambiguous import a warning message will be generated.

```
module Exp/AExp[Syntax]
  sort AExp .
  var AE[0-9'a-zA-Z]* : AExp .
end module

module Exp/AExp/Num[Syntax]
  imports Int, Exp/AExp .
  subsort Int < AExp .
end module

module Exp/AExp/Name[Syntax]
```

WADT'08, LNCS 5486, pp 187-205. 2009


```
  imports Exp/AExp .
  requires sort Name .
  subsort Name < AExp .
end module

module Exp/AExp/Plus[Syntax]
  imports Exp/AExp .
  _+_ : AExp AExp -> AExp .
end module

module Exp/BExp[Syntax]
  sort BExp .
  var BE[0-9'a-zA-Z]* : BExp .
end module

module Exp/BExp/Bool[Syntax]
  imports Bool, Exp/BExp .
  subsort Bool < BExp .
end module

module Exp/BExp/LessThanEq[Syntax]
  imports Exp/AExp, Exp/BExp .
  _<=_ : AExp AExp -> BExp .
end module

module Exp/BExp/Not[Syntax]
  imports Exp/BExp .
  not_ : BExp -> BExp .
end module

module Exp/BExp/And[Syntax]
  imports Exp/BExp .
  _and_ : BExp BExp -> BExp .
end module

module Stmt[Syntax]
  sort Stmt .
end module

module Stmt/Sequence[Syntax]
  imports Stmt .
  _;_ : Stmt Stmt -> Stmt .
end module

module Stmt/Assign[Syntax]
```

WADT'08, LNCS 5486, pp 187-205. 2009


```
  imports Stmt, Exp/AExp .
  requires sort Name .
  _:=_ : Name AExp -> Stmt .
end module

module Stmt/IfThenElse[Syntax]
  imports Stmt, Exp/BExp .
  if_then_else_ : BExp Stmt Stmt -> Stmt .
end module

module Stmt/While[Syntax]
  imports Stmt, Exp/BExp .
  while_do_ : BExp Stmt -> Stmt .
end module

module Stmt/Halt[Syntax]
  imports Stmt, Exp/AExp .
  halt_ : AExp -> Stmt .
end module

module Pgm[Syntax]
  imports Stmt, Exp/AExp .
  sort Pgm .
  _;_ : Stmt AExp -> Pgm .
end module
```

Using the abstract syntax, a number of modules are used to define the evaluation semantics, with one semantics module per language feature. As a reminder, a semantics module will automatically import the syntax module of the same name; explicit imports of syntax modules are used in cases where attributes, such as stricness, need to be changed. Added strictness information will cause heating and cooling rules to be automatically generated. The `seqstrict` attribute, used on less than, is identical to `strict`, except it enforces a left to right evaluation order on the arguments.

```
module Exp/AExp/Name[Dynamic]
  requires sort Name, sort Loc, sort Value .
  imports Env, Store, K/K .
  rl k(| [X ==> V] |> env<| (X,L) |> store<| (L,V) |> .
end module

module Exp/AExp/Plus[Dynamic]
  imports Exp/AExp/Plus[Syntax]
     with { op _+_ now strict, extends + [Int * Int -> Int] } .
end module
```

WADT'08, LNCS 5486, pp 187-205. 2009


```
module Exp/BExp/LessThanEq[Dynamic]
  imports Exp/BExp/LessThanEq[Syntax]
    with { op _<=_ now seqstrict, extends <= [Int * Int -> Bool] } .
end module

module Exp/BExp/Not[Dynamic]
  imports Exp/BExp/Not[Syntax]
    with { op not_ now strict, extends not [Bool -> Bool] } .
end module

module Exp/BExp/And[Dynamic]
  imports Exp/BExp/And[Syntax]
    with { op _and_ now strict(1) } .
  rl true and BE => BE .
  rl false and BE => false .
end module

module Stmt/Sequence[Dynamic]
  eq S ; S' = S ~> S' .
end module

module Stmt/Assign[Dynamic]
  imports Stmt/Assign[Syntax]
    with { op _:=_ now strict(2) } .
  requires sort Name, sort Value, sort Loc .
  imports K/K, Env, Store .
  rl k(| [ X := V ==> . ] |> env<| (X,L) |> store<| (L,[_ ==> V]) |>
end module

module Stmt/IfThenElse[Dynamic]
  imports Stmt/IfThenElse[Syntax]
    with { op if_then_else_ now strict(1) } .
  imports Bool .
  rl if true then S else S' => S .
  rl if false then S else S' => S' .
end module

module Stmt/While[Dynamic]
  imports Stmt/IfThenElse, Exp/BExp[Syntax] .
  eq k(| while BE do S |> = k(| if BE then (S ; while BE do S) else . |> .
end module

module Stmt/Halt[Dynamic]
  imports Stmt/Halt[Syntax]
    with {op halt_ now strict } .
```

WADT'08, LNCS 5486, pp 187-205. 2009


```
    imports Int .
    eq k(| halt i |> = k(| i |) .
  end module

  module Pgm[Dynamic]
    requires sort Value .
    eq top (| k(| V |) |> = V .
  end module
```

Finally, the various modules are assembled together into a language module, representing the entire programming language.

```
  module Imp[Language]
    imports K/Configuration, K/K, K/Location, K/Value,
            Env, Store, Int, Bool .
    config = top(store(Store) env(Env) k(K) nextLoc(Loc)) .

    op [[_]] : K -> Configuration .
    eq [[ K ]] = top(store(empty) env(empty) k(K) nextLoc(initLoc)) .

    imports type=Syntax Exp/AExp/Num, Exp/BExp/Bool .
    imports type=Dynamic Exp/AExp/Name, Exp/AExp/Plus,
      Exp/BExp/LessThanEq, Exp/BExp/Not, Exp/BExp/And,
      Stmt/Sequence, Stmt/Assign, Stmt/IfThenElse,
      Stmt/While, Stmt/Halt, Pgm .
  end module
```