Towards a Module System for K

Mark Hills and Grigore Roșu
{mhills, grosu}@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

WADT'08, 14 June 2008

Motivation

- Formal semantics of programming languages not used widely outside of research community
- We strongly believe that one important way to increase use of formal semantics is to make semantic definitions more broadly usable and useful
- This work focuses on two aspects of this goal:
    - Enable improved reuse of definitions through definitional modularity
    - Provide tool support for working with language definitions

K: High Level View

- K provides a rewrite-based method to formally define computation
- Focus here: formal definitions of programming languages
- Definitions should be flexible and modular: use, and reuse, for language documentation, program execution, analysis, proof

K Basics: Computations

- K based around concepts from Rewriting Logic Semantics, with some intuitions from Chemical Abstract Machines and Reduction Semantics
- Abstract *computational structures* contain context needed to produce a future computation (like continuations)
- Context can consist of lists or multisets, generally representing sequential or concurrent computation potential
- Context includes special component, `k`, made up of list of computational tasks separated by $\curvearrowright$, like $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$
- From here on, computational structures called computations

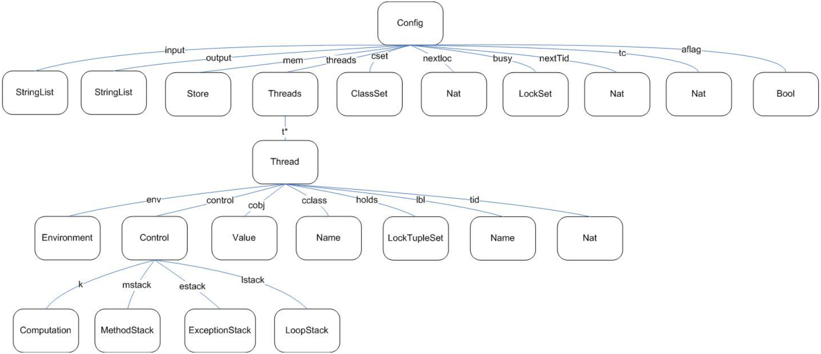
K Basics: Computations, Continued

- Intuition from CHAMs: language constructs can *heat* (break apart into pieces for evaluation) and *cool* (form back together)
- Represented using $\rightleftharpoons$, like $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$
- Operators containing $\square$ called *freezer*s
- Heating/cooling pair can be seen as an equation
- Intuition from RS: $\square$ can be seen as similar to evaluation contexts, marking the location where evaluation can occur

K Basics: Cells

- Computations take place in context of a *configuration*
- Configurations hierarchical (like in RLS), made up of K *cells*
- Each cell holds specific piece of information: computation, environment, store, etc
- Two regularly used cells:
  - $\top$ (*top*), representing entire configuration
  - k, representing current computation
- Cells can be repeated (e.g., multiple computations in a concurrent language)

Example: K Configuration



K Basics: Equations and Rules

- Computations defined used equations and rules
- Heating/Cooling Rules (Structural Equations): manipulate term structure, non-computational, reversible, can think of as just *equations*
- Rules: computational, not reversible, may be concurrent

Example: Equations

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$

$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$

$$\text{if } b \text{ then } s_1 \text{ else } s_2 = b \curvearrowright \text{if } \square \text{ then } s_1 \text{ else } s_2$$

Reminder: □ is not an evaluation context, but a freezer. Also, operations with freezers are boring to write, so we can mark operations `strict(natlist)`, with a freezer generated for each position in the list. To do so for all operands, just use `strict`.

```
_+_ : AExp AExp -> AExp [strict]
```

Example: Rules

$i_1 + i_2 \longrightarrow i$, where i is the sum of i_1 and i_2

if true then s_1 else $s_2 \longrightarrow s_1$

if false then s_1 else $s_2 \longrightarrow s_2$

$$(\!|\underset{\cdot}{\underline{x := v}}|\!\rangle_k \ \langle\!| (x, l) |\!\rangle_{env} \ \langle\!| (l, \underset{v}{\underline{\_}}) |\!\rangle_{store}$$

For More Information

For more information on K:

- “A Rewriting Logic Approach to Type Inference” (earlier talk)
- K website: `http://fsl.cs.uiuc.edu/k`
  - Includes tech reports and other papers related to K

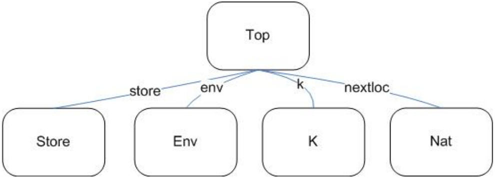
The Need for Context Transformers

- Rewriting logic semantics equations/rules (just rules from here, unless distinction matters) match across configuration items
- Configuration items provide *context* to where rule can apply

The Need for Context Transformers

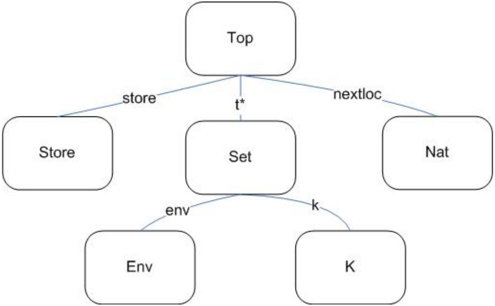
- Rewriting logic semantics equations/rules (just rules from here, unless distinction matters) match across configuration items
- Configuration items provide *context* to where rule can apply
- Problem: change in configuration structure can change context, break existing rules

Example



$$(\!| \underset{\cdot}{\underline{x := v}} \rangle_k \; \langle\!| (x, l) |\!\rangle_{env} \; \langle\!| (l, \underset{v}{\underline{\_}}) |\!\rangle_{store}$$

Example, After Configuration Change



$$\langle\!|\langle\!|\underset{\cdot}{\underline{x := v}}|\!\rangle_k \; \langle\!|(x, l)|\!\rangle_{env}|\!\rangle_t \; \langle\!|(l, \underset{v}{\underline{\_}})|\!\rangle_{store}$$

Context Transformers

- Context transformers solve problem by transforming context of rule to match configuration
- Handles almost all common cases using simple restrictions
  - Top level configuration items should have distinct names
  - Matching items are those closest together in graph
- Ambiguous matches will be flagged by tool
- User can explicitly specify context to handle unusual cases

Variable Patterns

- Provide way to define sorts of variables based on regular expressions for variable names
- Similar concepts found in other formalisms (e.g., ASF+SDF)
- Patterns visible throughout specification, not just in declaration module
- Example: `var Var[0-9]` for variables `Var0, Var1, ..., Var9`

Sort Inference

- In some specifications, variable declarations can make up half of spec
- Patterns help reduce this; *sort inference* can reduce even more
- Sorts of variables inferred based on definitions of ops
- Variables can always be explicitly declared or tagged with sorts if needed

$$\langle\!| \underset{\cdot}{\underline{x := v}} |\!\rangle_k \ \langle\!| (x, l) |\!\rangle_{env} \ \langle\!| (l, \underset{v}{\underline{\_}}) |\!\rangle_{store}$$

Module Formats

- Modules use similar syntax to Maude modules
- Module formats provided on top of standard module syntax to improve conciseness, allow defaults, enable special tool support
- Currently defined module formats: abstract syntax, language feature/semantics, configuration item, utility (similar to generic), language
- K provides built-ins (sets, maps, lists, etc); additional can be defined using standard algebraic techniques
- Note: Meta-information can be associated with all items in modules; not shown below to reduce clutter

Generic Modules

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

  sort Loc .
  sortalias Store = FiniteMap(Loc,Val) .
  subsort SSub < SSup .

  var V : Val . 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
```

Example: Abstract Syntax

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

- Undecorated module names are syntax modules
- Imports allow sort renaming
- Variable pattern declarations usable in other modules that (directly or indirectly) import this module
- Non-pattern variable declarations considered local

Example: Abstract Syntax

```
module Exp/AExp/Plus is
  imports Exp/AExp .
  _+_ : AExp AExp -> AExp .
end module
```

- Syntax defined using mixfix notation
- Any K attributes (strict, etc) defined directly on syntax considered defaults

Example: Semantics

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

- Type of dynamics given after path in [brackets]
- Any K attributes on syntax can be overridden on import
- Strictness auto-generates structural equalities for heating/cooling
- Extends uses predefined operations to give semantics to common constructs

Example: Semantics

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

- Strictness can be enforced on individual arguments; here only first strict for short-circuit evaluation
- Combination of rules and strictness assign meaning to language construct

Example: Semantics

```
module Exp/BExp/And[Static] is
  imports Exp/BExp/And with { op _and_ now strict } .
  rl bool and bool => bool .
  rl T1 and T2 => fail [ where T1 =/= bool or T2 =/= bool ] .
end module
```

- Strictness can be different for static or dynamic semantics
- Side condition added to distinguish `fail` case

Example: Language

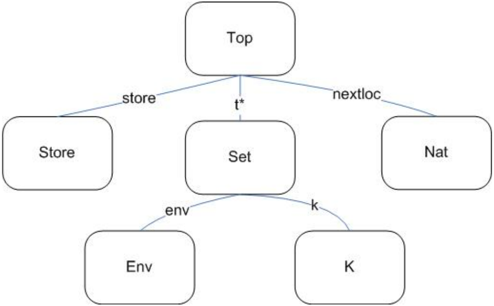
```
module Imp[Language] is
  imports type=Config Top, K, Env, Store .
  config = top(store(Store) env(Env) k(K) nextLoc(Nat)) .

  imports type=Dynamic
    Val/Int, Val/Bool, Exp/AExp/Name, Exp/AExp/Plus,
    Exp/BExp/LEq, Exp/BExp/Not, Exp/BExp/And, Stmt/Seq,
    Stmt/Assign, Stmt/IfThenElse, Stmt/While, Stmt/Halt, Pgm .
end module
```

- Language modules set up configuration, bring in semantics
- On import, type=tag syntactic sugar for Module[tag]
- `config` used to define state configuration

Config Example

```
config =
  top(store(Store)
    t*(env(Env) k(K))
    nextLoc(Nat)) .
```



Related Work, Briefly

- Modular Semantics: MSOS, Action Semantics, Monads, Modular ASMs (Montages)
- Rewriting Logic Semantics: work on modularity (Braga and Meseguer)
- Tool Support: Action Semantics, Montages, many others

Future Work

- Continue development of tools (Eclipse plugin, translation to Maude/K)
- Continue moving over language modules
- Online module database with links into tool: build languages through module sharing and reuse