Practical Techniques for Language Design and Prototyping

Mark-Oliver Stehr[1] and Carolyn L. Talcott[2]

[1] University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, IL 61801-2302, USA, `stehr@uiuc.edu`

[2] SRI International, Computer Science Laboratory, Menlo Park, CA 94025, USA, `clt@cs.stanford.edu`

Abstract. Global computing involves the interplay of a vast variety of languages, but practially useful foundations for language specification and prototyping at the semantic level are lacking. In this paper we present a systematic approach consisting of three techniques:

1. A generic calculus of explicit substitutions with names (called CINNI) that allows us to give a first-order representation of syntax to uniformly deal with all binding aspects.
2. An executable representation of Felleisen-style operational semantics in terms of first-order rewrite rules.
3. A logical framework, namely rewriting logic, that allows us to express (1) and (2) and, in addition, language aspects such as concurrency and non-determinism.

We illustrate the use of these techniques in two applications:

1. A formal specification and analysis of PLAN, a Packet Language for Active Networks, that has been developed in the Switchware project at UPenn. This work was conducted in the scope of the DARPA Active Network Program.
2. The development of CIAO, a Calculus of Imperative Active Objects, a core language for concurrent object-oriented programming. It is especially designed to allow the representation of practically relevant sublanguages of common object-oriented languages such as Java, C#, and C++. This second application is subject of ongoing work.

1 Introduction

Global computing poses many challenges for language design and semantics. The agreement on a single language for global computing application seems neither realistic nor desirable. Instead, global computing applications typically consist of interacting distributed components involving a variety of different languages. Such languages can be either general-purpose or domain-specific, they can be imperative or declarative, they can support a particular programming paradigm such as functional, logic-based, object-oriented, or agent-oriented programming, and in addition may support specialized or general paradigms to deal with concurrency, coordination, communication, and mobility.

As a consequence, language designers often face a complex multi-dimensional design space, and the understanding of the dynamics of systems critically depends on the semantics of the languages used in their development. Language design is influenced by many practical software engineering requirements and on the other side by theoretical considerations of simplicity or mathematical elegance. Hence, language design is largely an experimental and evolutionary activity that explores tradeoffs between diffferent design decisions that are not apparent without conducting at least prototype implementations of languages and case studies of systems developed using the prototypes. In this paper we propose a formal approach based on a combination of established techniques to support language design and prototyping, which should not only speed up the design cycle but at the same time provide a sound mathematical foundation for the semantics of the language being designed.

To this end, we combine the well-known Felleisen-style operational semantics, also known as *evaluation-context* or *reduction-context semantics*, with CINNI, a calulus of explicit substitutions that is *parametric* in the syntax of the object language, to obtain a formal description of the semantics of the object language as a set of *first-order rewrite rules* that operate *modulo an underlying equational theory.* The concrete logical framework that we are using in this paper is rewriting logic [38], more precisely the version that is based on an underlying membership equational logic [6] and implemented in the Maude engine [10, 11]. The executable semantics specification in rewriting logic then serves as a prototype implementation of the language and can be used not only to execute object programs in the traditional way, but it can also be used to perform several other forms of analysis such as symbolic execution, state space exploarion, and model checking.

The paper is organized as follows: We first introduce explicit substitutions and the CINNI calculus in Section 2, which will then be used in Section 3 to obtain an executable first-order representation of Felleisen-style semantics. In both sections we use a version of call-by-value λ-calculus to illustrate our approach. We will refer to this version of λ-calculus as our *base language*, because Sections 4 and 5 will deal with two very different applications which can both be regarded as extensions of this base language. The first application of our techniques is PLAN a domain-specific language for active networks. The second application is concerned with the design of CIAO, a general-purpose language for object-oriented concurrent programming.

2 CINNI - A Generic Calculus of Explicit Substitutions

For a uniform way to deal with binding we use the CINNI calulus of [46, 45]. CINNI is a generic first-order calculus of explicit substitutions that is parametric in the object language and that does not abstract away the names of variables. This is in contrast to representations of syntax modulo α-conversion, i.e. consistent renaming of bound variables, or representations based on de Bruijn indices [15]. In both of these representations the information about names is lost. In fact, by employing a notation based on indexed names that goes back to Berkling [5], CINNI generalizes de Buijn's notation and also the corresponding explicit substitution calculus $\lambda\upsilon$ of [34], in two dimensions, namely in the choice of the object language and in the notation for variables.

We say that the syntax of an object language is a CINNI syntax if there is a distinguished sort of identifiers and all variables, i.e. referencing occurrences of identifiers, are of the form X_i for an identifier X and an index $i \in \mathbb{N}$. The idea is that X_i refers to the X-binder that can be reached on the way towards the outermost position after skipping i X-binders. Hence, X_0 refers to the innermost encompasing X-binder. Given the CINNI syntax $\mathcal{L}$ of a language we use $CINNI_{\mathcal{L}}$ to denote the instantiation of CINNI to the syntax of $\mathcal{L}$. The language of $CINNI_{\mathcal{L}}$ extends $\mathcal{L}$ by simple substitutions $[X{:=}M]$, shift substitutions $\uparrow_X$, and lift substitutions $\Uparrow_X S$, and a notion of substitution application, written $S\ M$, assuming that S ranges over substitutions and M ranges over terms. The equations of $CINNI_{\mathcal{L}}$ given below define the semantics of these substitutions and have been shown to be strongly normalizing in [46, 45].

$$\begin{array}{rcl}
[X{:=}M]\ X_0 & = & M \\
[X{:=}M]\ X_{m+1} & = & X_m \\
[X{:=}M]\ Y_n & = & Y_n \text{ if } X \neq Y \\[6pt]
\uparrow_X\ X_m & = & X_{m+1} \\
\uparrow_X\ Y_n & = & Y_n \text{ if } X \neq Y \\[6pt]
\Uparrow_X S\ X_0 & = & X_0 \\
\Uparrow_X S\ X_{m+1} & = & \uparrow_X (S\ X_m) \\
\Uparrow_X S\ Y_n & = & \uparrow_X (S\ Y_n) \text{ if } X \neq Y
\end{array}$$

Furthermore, for each syntactic constructor f of $\mathcal{L}$ there is a *syntax-specific equation*

$$S\ f(P^1, ..., P^n) = f((\Uparrow_{P^{j_{1,1}}} ... (\Uparrow_{P^{j_{1,m_1}}} S))\ P^1, ..., (\Uparrow_{P^{j_{n,1}}} ... (\Uparrow_{P^{j_{n,m_n}}} S))\ P^n)$$

where $j_{i,1}, ..., j_{i,m_i}$ are all the arguments (necessarily of the sort of identifiers) that f binds in argument i (necessarily of the sort of expressions). If argument i of f is a binder, we define $(\Uparrow_{...} ... (\Uparrow_{...} S))\ P^i$ as P^i in the above equation. In this case P^i is of the form X for some identifier X and represents an X-binder (as opposed to a variable X_i), which is not subject to substitutions. This abuse of notation allows us to write the syntax-specific equations in the compact form given above.

The syntax-specific equations are the only equations that depend on the syntax of $\mathcal{L}$. For instance, $CINNI_\lambda$ has the following two syntax-specific equations, assuming we write $[X]\ M$ for λ-abstractions.

$$S\ (MN) \ = \ (SM)(SN) \qquad\qquad S\ ([X]\ M) \ = \ [X](\Uparrow_X S\ M)$$

Since substitutions can always be eliminated using these equations, each $CINNI_{\mathcal{L}}$ term reduces to a unique $\mathcal{L}$ term.

The base laguage that we use in this paper is an applied call-by-value λ-calculus, similar to what forms the basis of Scheme and ML (although we do not consider types in this paper). The abstract syntax, which we now introduce, uses CINNI notation for bound variables. Presupposing a sort `Nat` of natural numbers, and a sort of identifiers `Id` (which we assume contains quoted identifiers for the examples in this paper), variables (sort `Var`) are formalized as a subsort of expressions (sort `Ex`) as follows.

```
sort Var .
subsort Var < Ex .
op _{_} : Id Nat -> Var .
```

The elements of basic data types are modeled by injecting the corresponding Maude sort into the sort of constants (sort `Const`), which again is a subsort of expressions.

```
sort Const .
subsort Const < Ex .
op Bool_ : Bool -> Const .
op Int_ : Int -> Const .
op String_ : String -> Const .
op Dummy : -> Const .
ops Nil  : -> Const .
```

The functions associated with these built-in data types are classified into constructors (sort `Cstr`) and non-constructors (sort `NonCstr`). Some standard fuctions such as logical and arithmetic operators have been omitted here for the sake of brevity.

```
sorts Cstr NonCstr .
subsorts Cstr NonCstr < Const .
ops Pair : -> Cstr .
ops Cons  : -> Cstr .
ops Hd Tl : -> NonCstr .
```

Expressions (sort `Ex`) are built from constants and variables using typical functional and imperative language constructs shown below.

```
sort Ex .
op _(_) : Ex ExList -> Ex .
op If_Then_Else_ : Ex Ex Ex -> Ex .
op Lam`[_:_`]_ : IdList PlanTypeList Ex -> Ex .
op Let`[_=_`]_ : IdList ExList Ex -> Ex .
op LetRec`[_=_`]_ : IdList ExList Ex -> Ex .
op Skip : -> Ex .
op _;_ : Ex Ex -> Ex .
op Try_Catch`[_`]_ : Ex Id Ex -> Ex .
op Throw_ : Ex -> Ex .
```

The meaning of these constructs is the standard one of imperative call-by-value λ-calculus, but function application `_(_)` and λ-abstraction `` Lam`[_:_`]_ `` are generalized to arbitrary n-ary functions (so that currying is not needed), and correspondingly a single (recursive) `Let` construct allows several *simultaneous* bindings. Furthermore, our base language has a statement `Skip` without an effect, allows sequential composition `_;_` and supports standard constructs for handling and raining exceptions. Sequential composition will only be of use when expressions can have effects, as it will be the case in both extensions of the base language that we consider in Sections 4 and 5, respecticely.

Lists of identifiers (sort `IdList`) are generated by singletons (we identify an identifier and the corresponding singleton list) and by the concatenation constructor `_,_` with structural equations for associativity and identity (relative to the empty list `empty-idl`). List of other sorts such as expressions (sort `ExList`) are specified correspondingly.

```
sort IdList .
subsort Id < IdList .
op empty-idl : -> IdList .
op _,_ : IdList IdList -> IdList [assoc id: empty-idl] .
```

Below we have generalized the original CINNI substitutions to simultaneous substitutions by lifting substitutions from `Id` to `IdList` (which represents a simultaneous binding). There is the basic explicit substitution constructor `[_:=_]`, two auxiliary operations `shift` and `lift`, and an operation `__` for application of a substitution to an expression list (expressions being a special case).

```
sort Subst .
op [_:=_] : Id Ex -> Subst .
op [_:=_] : IdList ExList -> Subst .
op [shift_] : Id -> Subst .
op [lift__] : Id Subst -> Subst .
op [lift__] : IdList Subst -> Subst .
op __ : Subst Ex -> Ex .
op __ : Subst ExList -> ExList .

eq  [id := ex] id{0} = ex .                                     (E0)
eq  [id := ex] id{suc(m)} = id{m} .                             (E1)
ceq [id := ex] id'{m} = ex if id =/= id' .                      (E2)

eq  [shift id] id{m} = id{suc(m)} .                             (E3)
ceq [shift id] id'{m} = id'{m} if id =/= id' .                  (E4)

eq  [lift id S] id{0} = id{0} .                                 (E5)
eq  [lift id S] id{suc(m)} = [shift id] (S id{m}) .             (E6)
eq  [lift id S] id'{m} = [shift id] (S id'{m}) .                (E7)

eq  S const = const .                                           (E8)
eq  S (ex exl') = (S ex) (S exl') .                             (E9)
eq  S (Lam [idl] ex) = Lam [idl]([lift idl S] ex) .            (E10)
```

```
...

eq  [empty-idl := empty-exl] ex' = ex' .                          (E11)
ceq [(id,idl) := (ex,exl)] ex' =                                  (E12)
    [id := ex][idl := [shift id] exl] ex' if idl =/= empty-idl .

eq  [lift empty-idl S] ex' = (S ex') .                            (E13)
ceq [lift (id,idl) S] ex' =                                       (E14)
    [lift idl [lift id S]] ex' if idl =/= empty-idl .
```

Here and throughout the paper we have omitted variable declarations. If not explicitly stated, the sort of variables should be clear from the context or from the name of the variable. Note also that in addition to E8, E9, and E10, there are further syntax-specific equations which we have omitted here.

To give a flavor of how CINNI handles substitutions we show the reduction of a λ-application using the above equations and the following beta rule of λ-calculus.

```
rl (Lam [idl] ex) exl) => [idl := exl] ex .                       (beta)
```

We assume identifiers x and y for the following examples.

```
((Lam [x] (Lam [x] (x{0} x{1}))) x{0})
 => [x := x{0}] (Lam [x] (x{0} x{1}))                              beta
  = (Lam [x] [lift x [x := x{0}]] (x{0} x{1}))                      E10
  = (Lam [x] ([lift x [x := x{0}]] x{0}
              [lift x [x := x{0}]] x{1}))
  = (Lam [x] (x{0} [shift x]x{0}))                          E0,E1,E5,E7
  = (Lam [x] (x{0} x{1}))                                            E3
```

Here we can see that the original x{0} has become x{1} to maintain its reference to an external binding.

In the case of multiple arguments, substitutions consists of simultaenous bindings and can be applied to lists of expressions, as illustrated by another example below.

```
(Lam [x,y] (x{0} y{0})) (x{0},x{0})
 => [x,y := x{0},x{0}] (x{0} y{0})                                 beta
  = [x := x{0}][y := [shift x] x{0}] (x{0} y{0})                    E12
  = [x := x{0}][y := x{1}] (x{0} y{0})                               E1
  = [x := x{0}] ([y := x{1}] x{0} [y := x{1}] y{0})                  E9
  = [x := x{0}] (x{0} x{1})                                       E2,E0
  = ([x := x{0}] x{0} [x := x{0}] x{1})                              E9
  = (x{0} x{0})                                                   E0,E1
```

In practice, indices different from 0 appear rarely, but obviously their are essential for the correct internal operation of CINNI. Therefore, we use the convention of [46, 45] that the index 0 can be omitted. This can be easily formalized using additional equations that treat each referencing occurrence of an identifier X just like $X\{0\}$.

Finally, it is important to note that the `beta` rule given above will not be part of the semantics of our base langauge in this form. Using the approach of the next section it will be replaced by a more controlled version for the call-by-value λ-calculus.

3 Executable Felleisen-style Semantics

A simple and concise formalization of the semantics of a language simplifies our intuitive unsterstanding and facilitates mathematical reasoning about programs or about the semantics itself. Just as the CINNI calculus with its explicit names reduces the gap between the actual syntax and its formal representation, we are interested in a systematic approach to programming language semantics that reduces the gap between the actual program behavior and the rules representing it formally. Simulataneously, we aim at a semantics that is executable, but our primary concern is simplicity and elegence (as far as this is achievable for the real-world languages that we are concerned with) rather than effienciency of the execution.

In this paper we employ a *syntax-based approach* often called *evaluation-context* or *reduction-context semantics* [18, 35, 49], which simplifies the reduction machine and gives us a very direct connection between the (partially executed) program and the machine state. This approach uses *extended program syntax* to represent semantic entities. In particular, values are just a subset of expressions, and the control stack is implicitly represented by expressions with holes, called reduction contexts. Furthermore, the specification of languages is considerably simplified by formalizing environments as substitutions, thereby eliminating the need to treat environments explicitly. This is in contrast to, for instance, SECD machines, which carry the environment as an explicit component. To specify the abstract machine we use a general approach suitable for functional languages with side-effects which is based on [18, 28, 37].

Reduction contexts are a special form of contexts in which the holes correspond to positions where evaluation can take place. In the case of deterministic languages such as those treated in this paper, reduction contexts have a single hole and this hole is not in the scope of any binding operators, because potential binding operators have been eleminated by means of substitutions. Redexes correspond to machine instructions, they can be immediately reduced. In the pure call-by-value λ-calculus the redexes are λ-abstractions applied to values: $(\lambda id\ .\ ex)\ val$. In the richer base language they also include non-constructors applied to value lists and let expressions in which all bindings are value expressions. Mathematical descriptions of deterministic evaluation using reduction contexts are based on a unique decomposition lemma that says that an expression ex is either a value or it decomposes uniquely into a reduction context R and a redex r such that ex is the result of filling the hole in R with r (written $R[r]$) [18].

A typical transition rule in the reduction-context semantics for call-by-value λ-calus looks as follows. Two points are noteworthy: The rule is not a first-order rewrite rule, because there is an implict quantification of all reduction contexts.

Furthermore, it uses a standard notion of capture-free substitution on the right hand side.

$$R[(\lambda id\ .\ ex)\ val] \to R[ex[val/id]]$$

The substitution can be directly be translated into a CINNI explicit substitution, but an executable first-order representation of the quantifiaction over reduction contexts requires more effort. The solution we adopt is to replace the patterns $R[ex]$ by pairs (cx, ex) where cx is a first-order representation of R. In this way we make explicit the partitioning of a reducible expression into a context and a redex. Hence, the reduction state of the abstract machine is a pair, written `RedState`(cx, ex), consisting of a reduction context and the expression that is the current focus of reduction.

```
op  RedState : Cx Ex -> RedState .
```

The sort `Cx` contains expressions with any number of holes (including possibly none) in any position in which an expression could occur. Thus expression constructors are overloaded to construct contexts and there is an additional constant `?` to represent the hole:

```
sort Cx .
subsort Ex < Cx .
op `? : -> Cx .
op _(_) : Cx CxList -> Cx .
op Lam`[_`]_ : IdList Cx -> Cx .
...
```

The operation of hole filling is a special case of metavariable substitution (the hole being the only metavariable) and is generalized to allow filling of holes with contexts (context composition) and to apply to context lists (sort `CxList`), contexts being a special case. Hole filling is pure textual substitution and hence entirely straightforward, in contrast to the capture-free substitutions of CINNI, which we need to deal with binding contructs of the object language. The process of hole filling is formalized by the following operation.

```
op <`?`:=_>_ : Cx Cx -> Cx .
op <`?`:=_>_ : Cx CxList -> CxList .
eq < ? := cx > ? = cx .
eq < ? := cx > const = const .
eq < ? := cx >(cx' cxl) = (< ? := cx >cx')(< ? := cx > cxl) .
...
```

Recall that in the syntax specification we classified function symbols into constructors and non-constructors. This was done in order to identify the subset of the expressions that represent values. Roughly speaking, all constants are values and constructors applied to lists of values are values. A non-constructor applied to any list of expressions is a non-value requiring one or more steps of evaluation. Also, a constructor applied to a list containing a non-value is a non-value. As shown below, values and non-values are formalized as subsorts `Val < Ex` and `NonVal < Ex`, respectively.

```
sort Val NonVal ValList .
subsort Val < Ex .
subsort NonVal < Ex .
subsort Const < Val .
subsort EmptyExList < ValList < ExList .

subsort Val < ValList .
op _`,_ : ValList ValList -> ValList [assoc id: empty-exl] .

mb (Lam [idl : typel] ex) : Val .
mb (cstr vall) : Val .

mb (val(exl,nval,exl')) : NonVal .
mb (nval(exl)) : NonVal .
mb (ncstr(exl)) : NonVal .
mb ((Lam [idl] ex) exl) : NonVal .
mb (ex ; ex) : NonVal .
mb (If ex Then ex' Else ex'') : NonVal .
mb (Let [idl = exl] ex') : NonVal .
mb (LetRec [idl = exl] ex') : NonVal .
mb (Try ex Catch [id] ex') : NonVal .
```

Reduction machine rules The reduction machine maintains two invariants on `RedState`(cx, ex). (1) The cx component is a reduction context. (2) The entire program (in its current stage of evaluation) is given by `<? :=` ex`>` cx, i.e. by filling the hole in cx with the focus expression ex.

To get some intution for the operation of the reduction machine it is helpful to note that the inductive definition of the set of reduction contexts corresponds to peeling off basic reduction contexts one layer at a time until a redex is reached: $ex = R_0[....R_n[r]]$. These basic reduction contexts correspond to a control stack with R_n at the top. For example, the first layer of a function application $ex = val(vall, nval, exl)$, where $vall$ is a value list and $nval$ is a non-value expression, is the reduction context $R = val(vall, \texttt{?}, exl)$ expressing the left to right evaluation order semantics. Most of the action occurs at the inner basic reduction context (top of the stack). For example, suppose the above application fills the hole of an outer reduction context R' so that $ex' = R'[ex] = R'[R[nval]]$. When the evaluation of $nval$ leads to a value val' the hole is filled with that value, and the resulting expression is redecomposed if it still contains a redex. The new decomposition is parametric in the outer reduction context, that is, it has the form $R'[R''[r']]$ where $R''[r']$ is the unique decomposition of $R[val']$.

There are two kinds of reduction machine rules: *control rules* that move the focus to the next relevant redex; and *reduction rules* that perform the actual reductions. The control rules are derived from the notion of redex, and the reduction machine rules directly correspond to the original rules of the reduction context semantics. For function application, we have the following three control rules and the β-reduction rule.

```
rl  RedState(cx, nval(exl'))
    =>
    RedState(< ? := ?(exl') > cx, nval) .

rl  RedState(cx, val(vall', nval', exl'))
    =>
    RedState(< ? := val(vall', ?, exl') > cx, nval') .

crl RedState(cx, val)
    =>
    RedState(?, < ? := val > cx)  if cx =/= ? .

rl  RedState(cx, (Lam [idl] ex)(vall))
    =>
    RedState(cx, [idl := vall] ex) .
```

Starting from the focus on the entire program `RedState(?,ex)`, the first two control rules iteratively focusses on the next non-value expression till they reach the redex. After reducing the redex with the β-reduction rule, the focus moves back to the top of the entire program using the third control rule, and the process is repeated.

Optimized Reduction machine rules The naive version of the reduction machine involves many operations of hole filling and decomposition. A more efficient version uses the observation that the reduction context layers correspond to a stack and represents this stack using a lazy hole filling operator (without any equations).

```
op  <<`?`:=_>>_ : Cx Cx -> Cx .
op  <<`?`:=_>>_ : Cx CxList -> CxList .
```

The optimized control rules can now be written as follows:

```
rl  RedState(cx, nval(exl'))
    =>
    RedState(<< ? := ?(exl') >> cx, nval) .

rl RedState(cx, val(vall', nval', exl'))
    =>
    RedState(<< ? := val(vall', ?, exl') >> cx, nval') .

rl RedState(<< ? := cx >> cx', val)
    =>
    RedState(cx', < ? := val > cx)  .
```

As before the first two control rules move the focus to the next unevaluated argument in a function application. The third rule moves the current focus towards the top (viewing the program as a tree) if the current focus is a value, but not necessarily to the root as in the naive version. It is the only rule that uses the eager version of context hole filling. A very similar optimization has been proposed under the name "refocusing" in the functional implementation of interpreters [12]. We will continue to work with this optimized style throughout the remainder of this paper.

Further Reduction Machine Rules A representative selection of the remaining reduction machine rules for our base language is given in the remaining part of this section. First, the semantics of function application needs to be completed. There are reduction rules for the application of all built-in functions, the following just being one example.

```
rl  RedState(cx, Fst(Pair(val,val')))
    =>
    RedState(cx, val) .
```

In addition we have the straightforward control and reduction rules for the conditional construct.

```
rl  RedState(cx, (If nval Then ex' Else ex''))
    =>
    RedState(<< ? := (If ? Then ex' Else ex'') >> cx, nval) .

rl  RedState(cx, (If (Bool true) Then ex' Else ex''))
    =>
    RedState(cx, ex') .

rl  RedState(cx, (If (Bool false) Then ex' Else ex''))
    =>
    RedState(cx, ex'') .
```

Sequential composition of statements is specified by the following rules:

```
rl  RedState(cx, (nval ; ex'))
    =>
    RedState(<< ? := (? ; ex')>> cx, nval) .

rl  RedState(cx, (val ; ex'))
    =>
    RedState(cx, ex') .

rl  RedState(cx, (Skip ; ex'))
    =>
    RedState(cx, ex') .
```

There are control and reduction rules for the `Let` construct, which exactly as in the case of β-reduction is simply turned into an explicit substitution.

```
rl  RedState(cx, Let [idl = (vall', nval', exl')] ex'')
    =>
    RedState(<< ? := Let [idl = (vall', ?, exl')] ex'' >> cx, nval') .

rl  RedState(cx, Let [idl = vall'] ex'')
    =>
    RedState(cx, [idl := vall'] ex'') .
```

Then there are control and reduction rules for `LetRec`, where it is important to note that the scope of the expressions on the right hand side of `=` includes *all* variables introduces by the construct itself. For an elegant formulation we use a version of `LetRec` lifted to lists of expressions.

```
rl  RedState(cx, LetRec [idl = (vall', nval', exl')] ex'')
    =>
    RedState(<< ? := LetRec [idl = (vall', ?, exl')] ex'' >> cx, nval') .

op  LetRec`[_=_`]_ : IdList ExList ExList -> Ex .
eq  (LetRec[idl = exl] empty-exl) = empty-exl .
ceq (LetRec[idl = exl] (ex',exl')) =
    ((LetRec[idl = exl] ex'),(LetRec[idl = exl] exl'))
    if exl' =/= empty-exl .

rl  RedState(cx, LetRec [idl = vall'] ex'')
    =>
    RedState(cx, [idl := (LetRec [idl = vall'] vall')] ex'') .
```

Somewhat different than the previous rules are the rules for raising and handling exceptions. First, there is the control rule and the reduction rule for the non-error case:

```
rl  RedState(cx, (Try nval Catch [id] ex'))
    =>
    RedState(<< ? :=  (Try ? Catch [id] ex') >> cx, nval) .

rl  RedState(cx, (Try val Catch [id] ex'))
    =>
    RedState(cx, val) .
```

The error case, however, if it cannot be directly handled by the first rule below, requires propagation of `Throw` upward in the program till the either the exceptions can be directly handled or the program terminates with this exception.

```
rl  RedState(cx, (Try (Throw eval) Catch [id] ex'))
    =>
    RedState(cx, [id := eval] ex') .

rl  RedState(cx, ((Throw eval) exl))
    =>
    RedState(cx, (Throw eval)) .
```

```
rl  RedState(cx, (val (vall,(Throw eval),exl)))
    =>
    RedState(cx, (Throw eval)) .

rl  RedState(cx, (Throw eval) ; ex)
    =>
    RedState(cx, (Throw eval)) .

rl  RedState(cx, (If (Throw eval) Then ex Else ex'))
    =>
    RedState(cx, (Throw eval)) .
```

4 PLAN - A Packet Language for Active Networks

As an application of the techniques introduced above, we briefly summarize our work [47] on the formal specification of PLAN, a *Packet Language for Active Networks*. The presentation is simplified a bit, e.g. we have omitted types, several built-in language constructs and services, because its purpose in this paper is to serve as a illustration of our techniques for language design and prototyping.

Active networks are networks with nodes that do not operate according to a fixed scheme (e.g. as conventional routers) but are instead fully programmable and provide *execution environments* for programs that can be received from other nodes via the network. Active networks can be wired, wireless or hybrid networks. One may think of active networks as a generalization of conventional networks and as a step toward greater flexibility: Packets, which are *interpreted* by routers in conventional networks following rigid schemes, become programs, which are *executed* in active networks in a universal fashion. See [50] for a survey of active network research and the recent DARPA conferences on this subject [13, 14].

PLAN [25, 24, 41, 26, 33], is an imperative functional language similar to ML, but has a number of additional features, such as remote function execution and resource awareness. *Remote function execution*, means that functions can be invoked in such a way that the execution does not take place locally but in the execution environment of a different network node. To this end, the function call is treated as a so-called chunk, i.e. as a piece of data, which is transmitted to the destination node by means of a packet. *Resource awareness* refers to a mechanism which keeps track of computational resources and ensures that all PLAN programs are terminating. In addition, PLAN programs interact with their host nodes through *service package interfaces*. Basic services include provision of information about local network topology, local node properties, time, and routing. Other possible services include resident data services for (time-limited) data storage and retrieval.

Our sources for the informal semantics of PLAN included (in addition to conversations with members of the Switchware team) the PLAN specification document [31] and the paper [33] (a fairly detailed description of an operational semantics), an abstract version of PLAN for reasoning about security [32], and the PLAN programmers guide [27]. We have specified a more general language that we call the extended PLAN Language (briefly xPLAN). It is based on the full call-by-value λ-calculus and unrestricted recursion, whereas the functional core language of PLAN is similar to a first-order fragment of ML, but only allows a form of bounded recursion. This generalization leads to a syntactially simpler, more elegant model with many interesting possibilities for mobile code. The official PLAN language maps naturally to a subset of xPLAN defined by simple syntactic restrictions. The main restriction, which ensures termination of PLAN (and corresponding xPLAN) programs, is that recursive calls can only occur inside chunks, and the local or remote invocation of a chunk reduces the computational resources available by at least one unit. Furthermore, forwarding a packet to the next hop consumes one unit so that the standard hop counter scheme to avoid nontermination of routing is subsumed by this concept.

Our specification is organized in three main parts: syntax; network; and semantics. The syntax part is a fairly direct formalization in Maude of the syntax of xPLAN as an algebraic data type. The network part models basic network concepts such as locations, addresses, connections, and routing, with the minimal detail needed for the PLAN specification. The semantic part is the heart of the matter. The *multilevel concurrency* of active networks is very directly reflected in the computation state which is structured to provide clear boundaries for the scope of effects and information access.

- A network configuration is modeled as a multiset containing nodes and packets.
- With each node we associate a multiset of processes local to the node, which serve as execution environments for programs and can themselves execute concurrently within the node.
- Each process encapsulates the local state of the execution environment together with an abstract reduction machine.

4.1 Syntax

The syntax of xPLAN is an extension of the syntax of the call-by-value λ-calculus base language of Section 3. We presuppose a sort `Addr` of host addresses, which are not necessarily unique for a given host, because each host can have several network devices and each of these has an associated host addresss. Host addresses and keys for the resident data services are the constants that we are adding to the base langauge.

```
op Addr_ : Addr -> Const .
op Key_ : Int -> Const .
```

There is an additional constructor `Chunk` for chunks of code.

```
op Chunk : -> Cstr .
```

Furthermroe, there are constants for each of the service functions. Some examples are given below.

```
ops GetRB GetSource GetSrcDev : -> NonCstr . *** Proc. level
ops ThisHostIs GetNeighbors  : -> NonCstr .  *** Node level
ops OnNeighbor OnRemote : -> NonCstr .    *** Packet creation
ops Exists Get Put  : -> NonCstr .        *** Data repository
```

The service calls `GetRB`(), `GetSource`(), and `GetSrcDev`() are used to access information about the current process, namely the remaining amount of computational resources, the address of the originating host, and the address of the network device at which the packet arrived that initiated the current process. The service `ThisHostIs`(a) checks whether a given address a refers to a network device local to the current note, and `GetNeighbors`() returns the list of neighbors of the current node. `OnNeighbor`($chunk, dest, int, dev$) invokes the given chunk *chunk* at a neighbor *dest* using *dev* as the outgoing device and passes on *int* of its resource units for sending the packet containing the chunk and for its execution on the remote node. `OnRemote` is similar but allows execution on arbitrary nodes and hence may involve packet routing by means of a routing function that has to be passed as an additional argument. Finally, `Exists`(str, i), `Get`(str, i), and `Put`(str, i, val, exp) provide access to a resident data dictionary local to the current node, (str, i) being a composite access key, *val* the value to be stored, and *exp* the time till expiration.

4.2 Semantics

The semantics of xPLAN correspondingly extends the semantics of our base language. In the following we first explain how the global active network state is represented. Then, we discuss the additional transition rules specific to xPLAN by giving a few representative examples.

The global state of an active network is a configuration modeled as multiset whose elements are nodes, processes, packets, data sets, and a unique global key. The sort and constructor declarations are as follows. We assume sorts `Addr`, `Loc`, `Connection`, `Route` of host addresses, locations, connections (i.e. pairs of the form *src* `>>` *dest*), routes (i.e. pairs of the form *dest* `via` *con*, meaning that *dest* can be reached via the connection *con*), and sorts `AddrList`, `Connection`, `List`, `RouteList` of corresponding lists.

```
sort Configuration .
sort Node Packet Process FreshKey Data DataItem .
subsorts  Node Packet Process FreshKey Data < Configuration .
op empty-conf : -> Configuration .
op __ : Configuration Configuration -> Configuration
        [assoc comm id: empty-conf] .
op Node : Loc AddrList ConnectionList RouteList -> Node .
op Packet : Addr Addr Addr Int Int Const
            Val ValList -> Packet .
```

```
op Process : Loc Addr Addr Int Int RedState -> Process .
op FreshKey : Int -> FreshKey .
op Data : Loc DataItemList -> Data .
op DataItem : String Int Val Int -> DataItem
```

A network node has the form `Node`$(l, devs, nbrs, rt)$. The location l serves as its identifier, *devs* lists its network devices, *nbrs* gives the connections to neighbors, and *rt* is the node's routing table. The network topology is given by the combined device and neighbors information of all of its nodes.

A packet in transit has the form `Packet`$(dest, fdest, orign, ssn, rb, rf, val, vall)$, where *dest* specifies the next hop destination address on its route to the final destination *fdest*. Each packet has an originating packet, injected into the network by some application and has assigned a unique session key. *ssn* is the session key of the originating packet, and *orign* is the address of the originating application. *rb* is the amount of computational resources available to the packet for its execution, and *rf* is the packet's preferred routing function. The final two arguments make up a chunk with function *val*, and (evaluated) arguments *vall*.

A process has the form `Process`$(l, orign, ardev, ssn, rb, rs)$. The process was created when a packet with node l as its final destination arrived. The address *ardev* refers to the device at which the packet entered the node, *orign*, *ssn*, are the same as in the packet, *rb* is the remaining amount of computational resources, and *rs* is the reduction machine state (see below).

Admissible configurations have a single object of the form `FreshKey`(key) used to generate fresh keys for sessions and controlled data sharing. The integer *key* is incremented each time a key is generated.

For the resident data services each node `Node`$(l, \ldots)$ has an associated data object `Data`(l, dil) where *dil* is a list of data items. Data items have the form `DataItem`(id, k, val, ttl), where (id, k) constitutes a composite key under which the value *val* is stored. The last argument *ttl* determines the time until expiration of the data item (present for future compatibility, since time advance is currently not modeled).

The configuration evolves by means of local reduction machine rules and service rules. The local reduction machine rules are precisely the reduction machine rules of our base language. The service rules are further split into process, network, packet, and data service rules. We now give a few representative examples of such rules.

Process service rules use information held in the process but outside the reduction machine state. For example, application of `GetRB` returns the resource bound, i.e. the remaining computational resources, of the current process.

```
rl  Process(l, orign, ardev, ssn, rb,
      RedState(cx, (GetRB empty-exl)))
    =>
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Int rb))) .
```

Network service rules use the nodes local network information. For example, the service function `ThisHostIs` checks whether a given address is one of the nodes network devices.

```
rl  Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (ThisHostIs (Addr a))))
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Bool (contains(devs,a))))) .
```

Data service rules manipulate the nodes resident data storage. For example, the service function `Put` adds or updates a data item.

```
rl  Data(l,dil)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Put ((String str),(Key key),
                         val,(Int ttl)))))
    =>
    Data(l,put(dil,str,key,val,ttl))
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, Dummy)) .
```

Packet rules include rules for *emitting*, *delivering*, and *routing* packets in transit. The PLAN construct `OnNeighbor` is one of the two possibilities to initiate a remote function call which is given by a chunk `Chunk`($val, vall$). As we can see below, the execution of `OnNeighbor` leads to the emission of a packet which encapsulates this chunk.

```
crl Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (OnNeighbor ((Chunk (val,vall)),
                      (Addr dest),(Int int),(Addr dev)))))
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, (rb -int),
      RedState(cx, Dummy))
    Packet(dest, dest, orign, ssn, (int - 1), NoRoute,
           val, vall)
    if connection(devs,nbrs,(dev >> dest)) and
       (rb >= int) and (int > 0) .
```

Notice that the current amount of resources `rb` of the executing process is decreased by the amount given to the emitted packet, and that amount is then decreased by one corresponding to the use of one unit for the first hop. The routing function component of the packet is set to an irrelevant constant `NoRoute` above, because `OnNeighbor` can only send packets to immediate neigbors. The more general `OnRemote` service allows remote invocation on arbitrary locations and allows the user to specify a routing function which is passed along in the packet.

When a packet reaches its destination (next hop agrees with final destination) a process is created to evaluate the contained chunk.

```
crl Node(l,devs,nbrs,rt)
    Packet(dest, fdest, orign, ssn, rb, rf, val, vall)
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, dest, ssn, rb,  RedState(?,(val vall)))
    if (dest == fdest) and contains(devs,dest) .
```

As we can see from the last two rules, the use of CINNI yields an elegant solution to the subtle problems of binding and environment handling in the context of recursive remote function calls. There is no need the carry explicit environments in chunks, because all variables that are in the scope of binding constructs of xPLAN are bound by explicit substitutions if evaluation reaches the construct and hence will be eliminated by the equations of CINNI.

In addition to the rules presented above, there are rules for other services, rules to route packets not yet at their destination, and termination rules to remove processes that have completed their task.

4.3 Example

As a concrete example of a PLAN program, we will use one of the route finding programs published in [33]. The Maude representation of this program is shown below. The program has two main functions: `find`, which does a forward search for the node with the destination address, and `goback`, that returns to the source by the inverse route and `Print`s the route found. The forward search, like Hansel and Gretel, drops crumbs to mark the way back, by storing at each node visited a backpointer, i.e. the address of the network device it used when leaving the previous node. When a packet containing an invocation `find-prog-2`(*dest*) is injected at some node in the network with a given destination address *dest*, the computation is initialized by determining the address of the starting node (using the `GetSource` service), by generating a fresh key for labeling data (using the `GenerateKey` service), and by an initial call of the `find` function with this information. The network is then flooded with packets which propagate themselves from nodes that have not been previously visited. To this end, the `find` function first uses the resident data service `Exists` to check if an entry associated with the current key exists in the local dictionary of the current node. If this is not the case, the node has not been previously visited. Hence a new entry in the local dictionary under the same key is created using `Put` to store the backpointer. Next it is checked using `ThisHostIs` if the destination has been reached, and if this is the case the route is reported back to the source by calling `goback`, which recursively follows the backpointers until the source is reached and the route can be `Print`ed, which is assembled on the way. Otherwise, the auxiliary function `sendchild` is called in the body of `find` for each neighbor (using `Foldr` to iterate over the list of neighbors), and `sendchild` itself recursively invokes `find` on the given neighbor's address using the `OnNeighbor` construct. The remaining computational resources are equally distributed among all neighbors (the corresponding amount is computed in `childrb`). For a more detailed discussion and analysis of this program and its somewhat surprising behavior we refer to [48].

```
LetRec ['goback = Lam ['k,'route]
   If (ThisHostIs (GetSource empty-exl))
   Then (Print 'route)
   Else (Let ['nexthop = (Get ((String ""),'k))]
         Let ['d = (GetDevToHost 'nexthop)]
         Let ['newroute = (Cons ('d,'route))]
           (OnNeighbor ((Chunk ('goback, ('k,'newroute))),
                        'nexthop, (GetRB empty-exl), 'd)))]

LetRec ['find = Lam [('dest,'previous,'k)]
   If (Exists ((String ""),'k))
   Then Dummy
   Else ((Put ((String ""), 'k, 'previous, (Int 200)));
     If (ThisHostIs 'dest)
      Then ('goback ('k,Nil))
      Else (Let ['neighbors = (GetNeighbors empty-exl)]
            Let ['srcdev =  (GetSrcDev empty-exl)]
            Let ['childrb = ... ] *** divide up rb
            Let ['sendchild =  Lam ['n,'u] *** emit a find packet
               (OnNeighbor ((Chunk ('find,
                             ('dest,(Snd 'n),'k))),
                (Fst 'n), 'childrb, (Snd 'n)))]
            (Foldr ('sendchild,'neighbors,Dummy))))]

  ('find ((Addr dest), (GetSource empty-exl), (GenerateKey empty-exl))) .
```

In summary, our specification fully captures the intent of the specifications [31] and [32], but has the benefit of being both formal and executable. Furthermore, we have illustrated in [47] how this specification can be used at very different levels [16] ranging from execution of test configurations, symbolic search, and model checking analysis to verification of general properties of programs and of the language itself [48].

Furthermore, our formal specification of the PLAN semantics clarifies a number of issues that remain vague or unsatisfactory in the original mathematical specification [31] such as: the scope of names and the notion of binding (in particular in connection with recursive programs), the handling of environments (especially when packets are shipped), the treatment of side-effects of iterators, the mechanism of exception handing, and the concurrent and distributed nature of packet execution. By treating a less restrictive language xPLAN the semantics was simplified without sacrificing the essential features of PLAN (the proper PLAN subset is characterized by a simple type system).

Last but not least, our specification captures the general idea of a *programming language for mobile computation* based on an imperative λ-calculus with features such as recursive function calls with a simultaneous change of location, a concept that is very different from the notion of remote procedure calls, which are static and synchronous in nature.

5 CIAO - A Calculus of Imperative Active Objects

CIAO is an ongoing attempt to develop a core language for concurrent object-oriented programming that is capable of representing practically relevant sub-languages of actual programming languages like Java, C#, and C++. Although some features have not been covered yet, the current version of CIAO serves as an interesting illustration of the general approach to language design and semantics proposed in this paper.

Many sucessful attempts to develop and study calculi for object-orientation exist in the literature. There is an entire line of research on λ-calculus representations of object-oriented languages [23, 8, 30, 42] and the development of special calculi for object-oriented prgramming such as Abadi and Cardelli's ς-calculus [3]. In addition, a number of Java-like core languages have been used in the literature for different purposes, Classic Java [20], Featherweight Java [29], and Middleweight Java [21] being just some examples. Since most work in this area is focussed on the challange of types systems for object-oriented programming, only a few approaches deal with calculi for concurent object-oriented languages, notable exceptions being [39], [17], [22], [9]. In contrast to most of these references, which introduce a calculus to study particular aspects, our objective is rather modest and pragmatic, in the sense that we are interested in a calculus with a small representational distance to existing practical languages, and the main emphasis is on an easy-to-understand syntax-based presentation of its operational semantics following the systematic, executable, and formal approach introduced in this paper.

The main inspiration for CIAO comes from the imperative version of Abadi and Cardelli's ς-calculus [3], but we deviate from it and extend it in a few important points. First, we do not aim at minimality and simply define CIAO as an extension of our call-by-value λ-calculus base language with standard programming constructs. Similar to the ς-calculus, objects are represented as a set of labeled attributes, which represent either fields or methods. While methods can be seen as *passive code*, objects in CIAO can also carry *active code*, that is code that is executed while the object is in existence. Furthermore, objects in CIAO are intended to represent both classes and their instances, and the calculus assumes a particular organization, namely a two dimensional hierarchy of *classification* and *inheritance*. In the first dimension, each non-class object has a distinguished attribute labelled `class`, which refers to the class object it is an instance of. The second dimension expresses inheritance, more precisely single inheritance in the current version of CIAO. To this end, each object has an attribute labelled `super` which refers to the object that it extends. Hence, an object in the traditional sense corresponds to a chain of partial objects in CIAO. Apart from this explicit two-dimensional organization, which is exploited by operations like method invocation, another noteworthy deviation from the ς-calculus is that a means for method update does not exist, the justification being that this feature is not present in the established real-world programming languages we want to model.

5.1 Syntax

The syntax of CIAO extends the syntax of our base language as follows. We first add some reserved identifiers. The identifiers `self` and `this` are usually used for the formal self parameter of a class and the formal this parameter of each method, although this convention is not a strict requireent. The identifier `unused` will be used only to introduce some syntactic sugar and is by convention never referenced.

```
ops super class new value self this unused : -> Id .
```

We also add constants denoting references. We have a distinguished `Null` reference and references to objects stored in the heap are constructed using `<` *p* `>`.

```
op Null : -> Const .
op <_> : Nat -> Const .
```

We also add a standard iteration construct which was not included in our base language:

```
op While_Do_ : Ex Ex -> Ex .
```

Now there are the following additional constructs specific to concurrent object-oriented programs. We have an explicit syntax for an *active object*, namely `Object <` *sid* `:` *cid* `|` *aexl* `|` *mexl* `|` *cex* `>`. Here, *cid* is a class identifier, *aexl* and *mexl* are the lists of fields and methods, respectively, and *cex* is the active code associated with the object. The identifier *sid*, which is called the *formal self parameter* and will usually be denoted as `self`, is bound by this construct and can can be used in *mexl* and *cex* to refer to the object itself, e.g. to access the object's own attributes. If *aexl* or *mexl* is the emty list we simply omit it for better readability. Similarly, we omit the active code *cex* if it is just `Skip`. In fact, this is the special case of a *passive object* prevailing in most object-oriented programming languages today.

```
sorts AttrEx .
op _:_ : Id Ex -> AttrEx .
op Object`<_:_|_|_|_> : Id Id AttrExList AttrExList Ex -> Ex .
```

Objects can then be created on the heap using `New`, which returns a reference to the given object. Hence, the `New` construct can be used to create instances of classes, and hence will typically be called by a factory method `new` of the corresponding class object, but `New` will also be used to create boxed versions (object representations) of the elements of basic data types, which are stored in the heap and can be subject to modifications. In fact, the is uniform way to deal with variable assignments in CIAO.

```
op New_ : Ex -> Ex .
```

Given an object reference *ex*, we use *ex* `.` *id* to access the field labelled *id*, and we use *ex* `.` *id* `:=` *ex*′ to update the field with the value of *ex*′. Finally, we have a generalized method invocation *ex* `@` *ex*′ `.` *id* `(` *exl* `)`, which invokes the method *id* with arguments *exl* on the objects referenced by *ex*, but uses the object referenced by *ex*′ for dynamic binding, i.e. the determination of method code to be executed. Usually, *ex* and *ex*′ will refer to the same object, but there are common programming patterns where we explicitly wish to restrict dynamic binding, e.g. to methods of the `super` object.

```
op _._ : Ex Id -> Ex .
op _._=_ : Ex Id Ex -> Ex .
op _@_._`(_`) : Ex Ex Id ExList -> Ex .
```

Since we now need to work modulo an instance of CINNI for CIAO syntax-specific equations corresponsing to the above constructs are added in the systematic way layed out in Section 2.

5.2 Semantics

Along with the new syntactic constructs we also extend our classification of expressions into values and non-values, as shown below. It is noteworthy that in order to evaluate an object only its fields will be evaluated, but neither its methods (which by convention are λ-abstractions, and will not be evaluated before their are invoked) nor its active code (which will be not evaluated before the object is stored in the heap).

```
sort AttrVal AttrNonVal .
mb (id : val) : AttrVal .
mb (id : nval) : AttrNonVal .

mb (Object < sid : id | avall | mexl | ex >) : Val .
mb (Object < sid : id | (aexl,anval,aexl') | mexl | ex >) : NonVal .

mb (New ex) : NonVal .

mb (ex . id) : NonVal .
mb (ex . id = ex') : NonVal .
mb (ex @ ex' . id (exl)) : NonVal .
```

Correspondingly, we extend our hole filling operator `<?:=_>_` to the new syntactic constructs in the obvious way.

The next step is the definition of the global state. Since we are dealing with potentially interacting concurrent objects, a multiset is the most suitable representation. In order to generate objects using `New`, the global state needs to maintain information about the next available heap reference `Fresh(`p`)`. For the runtime representation of an object in the heap we use a syntax similar to the notation for objects in programs (sort `Object` below). The active code, however, is not only an expression, but an entire reduction machine state. We introduce an auxiliary function `setRedState` to update this component.

```
sort Fresh .
op Fresh : Int -> Fresh .

sort Object .
op <_:_|_|_|_> : Nat Id AttrExList AttrExList RedState -> Object .

op setRedState : Object RedState ~> Object .
eq setRedState(< p : cid | aexl | mexl | rstate >, rstate') =
               < p : cid | aexl | mexl | rstate' > .
```

A configuration (sort `Configuration`) is then formalized as a multiset over these two sorts, but more often we are interested in pure multisets of objects (sort `Objects`).

```
sort Objects Configuration .

subsort Object < Objects < Configuration .
subsort Fresh < Configuration .

op empty-conf : -> Objects .
op __ : Configuration Configuration -> Configuration
        [assoc comm id: empty-conf] .
op __ : Objects Objects -> Objects
        [assoc comm id: empty-conf] .
```

A global state (sort `Everything`) is not exactly a configuration, but uses a toplevel constructor `{{_}}` to denote an explcitly closed configuration, that is a configuration that contains all objects. This notion is needed, as we will see later, because the effect of operations on an object is not necessarily restricted to the object itself, but in principle may involve any object of the configuration.

```
sort Everything .

op {{_}} : Configuration -> Everything .
```

Iteration rule Now we a prepared to present the transition rules associated with each new syntactic construct of CIAO. We begin with the reduction rule for the iteration construct, which is simply defined in terms of its unfolding:

```
rl  RedState(cx, (While ex Do ex'))
    =>
    RedState(cx, (If ex Then (Do ex' ; (While ex Do ex')))) .
```

Object creation rules The construct `New(ex)` allows us to create a runtime object representing *ex* in the heap. The first two rules are the control rules. As we can see from the second rule, to evaluate an object it is sufficient to evaluate its fields. The third rule deals with the case where *ex* is an object expression. Here, the formal self parameter *sid* is instantiated with the reference to the object itself. Recall that *sid* is bound in the methods $mexl'$ and in the exctive code ex', but not in the fields $avall'$. This is consistent with the fact that the fields are evaluated before the object is stored, but the actual reference of *sid* becomes available only when the object is actually stored in the heap.

```
rl  RedState(cx, (New nval))
    =>
    RedState(<< ? := (New ?)>> cx, nval) .

rl  RedState(cx, (Object < sid : cid | avall', (id' : nval'), aexl'
                            | mexl | actex >))
    =>
    RedState(<< ? := (Object < sid : cid | avall', (id' : ?), aexl'
                                | mexl | actex >) >> cx, nval') .

rl  Fresh(p)
    < q : cid | aexl | mexl
        | RedState(cx, (New (Object < sid : cid' | avall'
                                        | mexl' | ex' >))) >
    =>
    Fresh(s p)
    < q : cid | aexl | mexl
        | RedState(cx, < p >) >
    < p : cid' | avall' | [sid := < p >] mexl'
        | RedState(?, [sid := < p >] ex') > .
```

The subsequent reduction rule takes care of the case where *ex* is an integer, which is encapsulated as an object when stored in the heap. Similar rules exist for other constants.

```
rl  Fresh(p)
    < q : cid | aexl | mexl | RedState(cx, (New (Int int))) >
    =>
    Fresh(s p)
    < q : cid | aexl | mexl | RedState(cx, < p >) >
    < p : 'Integer | value : (Int int) | none | nocode > .
```

Object-oriented programming languages support boxed and unboxed representations of atoms, both with the capability to modify the content. In practice, unboxed representations are more efficient, but boxed representations offer the advantage of a uniform object-oriented treatment of data. Unboxed representations of this kind can be understood as an optimization of boxed representations and hence are not incorporated into CIAO to avoid conceptual redundancy. CIAO, however, allows unboxed representations *without* the capability of modification to be used for the purpose of functional computation and to serve as an intermediate step in the construction of boxed representations using `New` as in the previous rule.

The following syntactic sugar can be introduced for explicit unboxing and the usual notation of variable assignment:

```
op unbox_ : Ex -> Ex .
op _:=_ : Ex Ex -> Ex .

eq (unbox ex) = ex . value .
eq (ex := ex') = (ex . value = ex') .
```

Field access and update rules Before we give the rules for operations on objects we first introduce a few auxiliary functions to access and modify fields and to access methods. The search for the matching label follows the inheritance chain given by the `super` field if it exists. Note that these are all partial functions, because the search maybe unsuccessful, and the membership operator `::` is used to check definedness. Below we use *obj* and *objs* to range over `Object` and `Objects`, respectively.

```
op super : AttrExList ~> Nat .
ceq super(aexl) = p if < p > := get(aexl,super) .

op get : AttrExList Id ~> Ex .
eq get(((id : ex),aexl), id) = ex .
ceq get(((id : ex),aexl), id') = get(aexl,id') if id =/= id' .

op getField : Objects Nat Id ~> Ex .
eq getField(< p : cid | aexl | mexl | rstate > objs, p, id) =
   if (get(aexl,id) :: Ex) then get(aexl,id)
                           else getField(objs, super(aexl), id) fi .

op getMethod : Objects Nat Id ~> Ex .
eq getMethod(< p : cid | aexl | mexl | rstate > objs, p, id) =
   if (get(mexl,id) :: Ex) then get(mexl,id)
                           else getMethod(objs, super(aexl), id) fi .

op set : AttrExList Id Ex ~> AttrExList .
eq set(((id : ex),aexl), id, ex') = ((id : ex'),aexl) .
ceq set(((id : ex),aexl), id', ex') = ((id : ex),set(aexl,id',ex'))
    if id =/= id' .

op setField : Objects Nat Id Ex ~> Objects .

eq setField(< p : cid | aexl | mexl | rstate > objs, p, id, ex') =
   if (set(aexl,id,ex') :: AttrExList)
   then < p : cid | set(aexl,id,ex') | mexl | rstate > objs
   else < p : cid | aexl | mexl | rstate >
          setField(objs, super(aexl), id, ex') fi .
```

Finally, we will need an auxiliary notion of a multiset of objects with a distinguished element, which we write as a pair of an object and all remaining objects. The function `setField` is lifted to this structure.

```
sort ObjectsPair .

op `(_`,_`) : Object Objects -> ObjectsPair .
op __ : Object ObjectsPair -> ObjectsPair .
eq obj (obj',objs') = (obj',obj objs') .

op setField : ObjectsPair Nat Id Ex ~> ObjectsPair .

eq setField((< p : cid | aexl | mexl | rstate >, objs), p, id, ex') =
   if (set(aexl,id,ex') :: AttrExList)
   then (< p : cid | set(aexl,id,ex') | mexl | rstate >, objs)
   else (< p : cid | aexl | mexl | rstate >,
           setField(objs, super(aexl), id, ex')) fi .

eq setField((obj, < p : cid | aexl | mexl | rstate > objs), p, id, ex') =
   if (set(aexl,id,ex') :: AttrExList)
   then (obj, < p : cid | set(aexl,id,ex') | mexl | rstate > objs)
   else < p : cid | aexl | mexl | rstate >
          setField((obj,objs), super(aexl), id, ex') fi [owise] .
```

The semantics of language constructs for accessing and updating fields can be now formulated as follows. First we give the control rules, and then the two reduction rules for field access (`<` *p* `>` `.` *id*) and field update (`<` *p* `>` `.` *id* `:=` *val*′) respectively. Using the toplevel constructor `{{_}}` we make sure that both of these rules have access to all objects in the heap. Both rules involve two steps, first the information in the object referenced by `p` is accessed or updated, and then the reduction machine state of the active object which executes the instruction is updated.

```
rl  RedState(cx, (nval . id))
    =>
    RedState(<< ? := (? . id)>> cx, nval) .

rl  RedState(cx, (nval . id = ex'))
    =>
    RedState(<< ? := (? . id = ex')>> cx, nval) .

rl  RedState(cx, (val . id = nval))
    =>
    RedState(<< ? := (val . id = ?)>> cx, nval) .
```

```
crl  {{ Fresh(m) obj objs }} => {{ Fresh(m) obj' objs }}
 if < q : cid | aexl | mexl | RedState(cx, < p > . id) > := obj  /\
    ex' := getField(obj objs, p, id) /\
    obj' := setRedState(obj, RedState(cx, ex')) .

crl  {{ Fresh(m) obj objs }} => {{ Fresh(m) obj'' objs' }}
 if < q : cid | aexl | mexl | RedState(cx, < p > . id = val') > := obj  /\
    (obj', objs') := setField((obj,objs), p, id, val') /\
    obj'' := setRedState(obj', RedState(cx, val')) .
```

Method invocation rules The remaining operation is method invocation. There are three control rules and then a reduction rule for `<` p `>` `@` `<` p' `>` `.` *id* `(` *vall* `)`. In the control rules it is critical for the correct semantics of method invocations (in the presence of exceptions) that the arguments are evaluated before the method lookup takes place. The reduction rule again proceeds in two steps. First, it looks up the method code, which is expected to be a λ-abstraction of the form `Lam [` *id*′, *idl*′ `]` *ex*. The method lookup will start at the object referenced by `<` p' `>`, which can be higher in the inheritance hierarchy than the the object referenced by `<` p `>` on which the method is invoked. The first formal parameter is called the *formal this parameter* and usually denoted by `this`. In the second step, this parameter is substituted by `<` p `>`, and the remaining formal parameters are substituted by the evaluated arguments *vall* of the method invocation.

```
rl  RedState(cx, (nval @ ex . id (exl')))
    =>
    RedState(<< ? := (? @ ex . id (exl')) >> cx, nval) .

rl  RedState(cx, (val @ nval . id (exl')))
    =>
    RedState(<< ? := (val @ ? . id (exl')) >> cx, nval) .

rl  RedState(cx, (val @ val' . id (vall', nval', exl')))
    =>
    RedState(<< ? := (val @ val' . id (vall', ?, exl')) >> cx, nval') .

crl {{ Fresh(m) obj objs }} => {{ Fresh(m) obj' objs }}
 if  < q : cid | aexl | mexl |
           RedState(cx, < p > @ < p' > . id (vall)) > := obj /\
     (Lam [id',idl'] ex) := getMethod(obj objs,p',id) /\
     obj' := setRedState(obj,
                  RedState(cx, (Lam [id' := < p >] [idl' := vall] ex))) .
```

Since in the most common case the location of dynamic binding and the object of method invocation are identical, we introduce the following syntactic sugar:

```
op _._(_) : Ex Id ExList -> Ex.
eq ex . id (exl) = ex @ ex . id (exl) .
```

In the object-oriented programming style, method calls are often used to communicate among objects using well-defined interfaces, which is why they are traditionally interpreted as messages that can be understood by their receivers. Although in truely concurrent setting it is very tempting to replace this notion of method invocation with an asynchronous message exchange between concurrent threads of control as for instance in [39], CIAO keeps the common sequential notion of method invocation, where the thread of the caller rather than the callee is evaluating the method body. The rationale for this descision is again the goal the reduce the gap to actual programming languages. Clearly, this does not exclude an asynchronous message-passing style where the only function of such a method invocation is the (simulated) transmission of a message.

5.3 Examples

In the remaing part of this section we give a few simple examples showing how standard features of object-oriented languages can be expressed in CIAO. Although our approach is not specific to Java, we use Java-like code fragments and give their representation in CIAO. Hence the examples also convey an idea of how a representation mapping from a Java-like language into CIAO could be defined.

We begin with a program introducing a single class `Cell` of memory cells which can hold integer values. The program generates two instances of this class and modifies them subsequently.

```
class Cell
{
  int val = 0;
  Cell() {}
  Cell(int init) { val = init; }
  int get() { return val; }
  void set(int newval) { val = newval; }
}

Cell cell1 = new Cell(5);
Cell cell2 = new Cell(6);

cell1.set(7);
cell2.set(cell1.get());
```

The representation in CIAO introduces a corresponding class object named `'Cell`, which serves as a factory for its object instances. The operator `new`, which is usually a built-in operator in object oriented languages, become an explicit method of the class object. The invocation of `new` creates an instance of the class with the default initialization. The default initialization is explicitly overwritten by the more specific class initializer `'Cell`. This behaviour is implicit in the constructor `Cell` of the above Java-like program. In other words, a non-default initializer which is part of the object contructor is always made explicit in our representation. Also note that the object of each field access is make explicit using the formal self parameter.

```
Let ['Cell =
  New Object < self : 'Class
   || new : Lam [this]
        New Object < self : 'Cell
         | 'val : Int 0
         | 'Cell : Lam [this,'init] self . 'val = 'init,
           'get : Lam [this] self . 'val,
           'set : Lam [this,'newval] self . 'val = 'newval |> |> ]

Let ['cell1 = 'Cell . new ()] 'cell1 . 'Cell (Int 5);
Let ['cell2 = 'Cell . new ()] 'cell2 . 'Cell (Int 6);

'cell1 . 'set (Int 7);
'cell2 . 'set ('cell1 . 'get ())
```

The next example deals with inheritance. The class `Cell` from above is extended by a counter which keeps track of the number of modifications of each instance of this class. The constructor `CountingCell` reuses the constructor `Cell` of the super class, as indicated by the invocation of `super`. Similarly, the method `set` in the new subclass `CountingCell` overwrites `set` in `Cell`, but reuses the old code using a method invocation qualified by `super`.

```
class CountingCell extends Cell
{
  int counter = 0;
  CountingCell(int init) { super(init); }
  void set(int newval)  { counter = counter + 1;  super.set(newval); }
}

Cell cell1 = new CountingCell(5);
Cell cell2 = new CountingCell(6);

cell1.set(7);
cell2.set(cell1.get());
```

In the CIAO representation we again create a class object for the new subclass with a `new` method to create instances. In this case each instance has a `super` field which is initialized with an instance of the superclass, i.e. an instance of `Cell`. The invocation of `super` in `CountingCell` is represented by following the inheritance chain given by the `super` field and explicitly invoking the initializer of the super class. The invocation of `super` in `set` has a different semantics. It is represented by invoking the `set` method of the super class on the object `this` on which `set` is invoked. Here it is essential that, the scope of dynamic binding is explicitly restricted to the super class (and everything above).

```
Let ['CountingCell =
  New Object < self : 'Class
   || new : Lam [this]
        New Object < self : 'CountingCell
         | super : 'Cell . new (),
           'counter : Int 0
         | 'CountingCell : Lam [this,'init]
              self . super . 'Cell ('init),
           'set : Lam [this,'newval]
              self . 'counter = self . 'counter + Int 1 ;
              this @ (self . super) . 'set ('newval) |> |> ]

Let ['cell1 = 'CountingCell . new ()] 'cell1 . 'CountingCell (Int 5);
Let ['cell2 = 'CountingCell . new ()] 'cell2 . 'CountingCell (Int 6);

'cell1 . 'set (Int 7);
'cell2 . 'set ('cell1 . 'get ())
```

For a more concise formulation of the creation of classes and the pattern of class extension we introduce some syntactic sugar:

```
op New`Class_<_|_|_|> : Id Id AttrExList AttrExList -> Ex .
op New`Class_Extending_<_|_|_|> : Id Ex Id AttrExList AttrExList -> Ex .

eq New Class cid < sid | aexl | mexl |> =
  New Object < self : 'Class
   || new : Lam[unused]
        New Object < sid : cid | aexl | mexl |> |> .

eq New Class cid Extending ex < sid | aexl | mexl |> =
  New Object < self : 'Class
    || new : Lam[unused]
         New Object < sid : cid | super : (ex . new ()), aexl | mexl |> |> .
```

Using these abbreviations the previous representation can be rewritten in a form that is very close to the original Java-like code:

```
Let ['Cell =
 New Class 'Cell < self
   | 'val : Int 0
   | 'Cell : Lam [this,'init] self . 'val = 'init,
     'get : Lam [this] self . 'val,
     'set : Lam [this,'newval] self . 'val = 'newval |> ]

Let ['CountingCell =
  New Class 'CountingCell Extending 'Cell < self
   | 'counter : Int 0
   | 'CountingCell : Lam [this,'init] self . super . 'Cell ('init),
     'set : Lam [this,'newval]
        self . 'counter = self . 'counter + Int 1 ;
        this @ (self . super) . 'set ('newval) |> ]
...
```

A slight variation of the previous example uses a single class variable (rather than an instance variable) as a counter the keep track of the number of modifications of the counter across all instances. The code is shown below.

```
class Cell
{
  static int counter = 0;
  int val = 0;
  Cell(){}
  Cell(int init) { val = init; }
  int get() { return val; }
  void set(int newval) { counter = counter + 1;  val = newval; }
}

Cell cell1 = new Cell(5);
Cell cell2 = new Cell(6);

cell1.set(7);
cell2.set(cell1.get());
```

To deal with class variables each instance explicitly maintains a `class` field, which is initialized by the formal self parameter of the class object (and not the formal self parameter of the instance). This is correctly represented below, because the second formal self parameter of the instance is *not* in the scope of the field section of an object. With the `class` field being available, all class variable references are explicitly qualified using `self . class`.

```
Let ['Cell =
  New Object < self : 'Class
   | 'counter : Int 0
   | new : Lam [this]
       New Object < self : 'Cell
        | class : self,
          'val : Int 0
        | 'Cell : Lam [this,'init] self . 'val = 'init,
          'get : Lam [this] self . 'val,
          'set : Lam [this,'newval]
             self . class . 'counter = self . class . 'counter + Int 1 ;
             self . 'val = 'newval |> |> ]

Let ['cell1 = 'Cell . new ()] 'cell1 . 'Cell (Int 5);
Let ['cell2 = 'Cell . new ()] 'cell2 . 'Cell (Int 6);

'cell1 . 'set (Int 7);
'cell2 . 'set ('cell1 . 'get ())
```

Finally, we give an example illustrating the use of active objects. All objects encountered until now are passive, that is their code section was empty, abbreviating the trivial statement `Skip`. Below, we use threads, which in a Java-like language are object of special class `Thread`, which abstracts from system-level details of thread implementation. Our example below is a trivial application of threads, but it is sufficient to convey the main idea. We introduce a class `ThreadExt` which extends `Thread` with a local thread state captured by the instance variable `x` and a piece of code defined by the method `run`. This code will be executed when the thread is started, using the method `start` which is inherited from `Thread`. Two trivial concurrent threads are created and started in the example below.

```
class ThreadExt extends Thread
{
  int x = 0;
  public void run()  { x = 1; }
}

ThreadExt thread1 = new Thread();
ThreadExt thread2 = new Thread();

thread1.start();
thread2.start();
```

Using the syntactic sugar introduced earlier, the CIAO representation of this program is as follows. First, the built-in `Thread` class needs to represented explicitly. The idea is that it encapsulates the thread state in an instance variable `state`, which is initialized with an active code upon invocation of `start`. The active object then invokes that actual code of the thread. This encapsulation into an additional active object is essential, because the active code of CIAO obejcts executes as soon as the object comes into existence, as opposed to the behavior of threads which can be created and then need to be started explicitly.[1]

```
Let ['Thread =
  New Class 'Thread < self
    | 'state : Null
    | 'run : Lam [this]  Skip,
        'start : Lam [this]
            self . 'state = New Object < self ||| this . 'run () > |> ]

Let ['ThreadExt =
  New Class 'ThreadExt Extending 'Thread  < self
    | 'x : Int 0
    | 'run : Lam [this] self . 'x = Int 1 |>]
```

[1] A detail that we glossed over here is that restarting the CIAO thread creates another active object. This is not relevant for the semantics of our example, but does not correspond to the Java semantics for general programs. It is easy to enhance our representation to handle details like this.

```
Let ['thread1 = 'ThreadExt . new ()]
Let ['thread2 = 'ThreadExt . new ()]

'thread1 . 'start ();
'thread2 . 'start ()
```

Just as in the case of PLAN, CIAO programs can be executed and analyzed using the tools offered by Maude. What is particularly appealing is that there is hardly any gap between the dynamic representation of the program and the program itself, so that program errors, if not explicitly handled, will simply leave the program in a state that directly corresponds to the remaining part of the program, and hence is easy to understand. Another interesting practical aspect of our representation of objects, is that it uses a syntax similar to the syntax for rule-based object-oriented programming [39] in Maude, which should facilitate the integration between Maude and languages supporting the classical style of object-oriented programming.

Finally, we should point out that there is a lof of room for the extension of our calculus. Common control flow constructs as known from C, C++, Java, C#, have already been added to CIAO, but we have intentionally not included them here to avoid too much syntactic overhead. However, there are more features, e.g. casting, arrays, synchronization, interfaces, multiple inheritance, which still need to be added in a reasonably clean way. Other features, e.g. the access to hidden instance variables, seems less important but could be supported if necessary. Typechecking, overloading, modules/packages and access control are features that should be treated at an earlier stage, i.e. are outside of the scope of the operational semantics. High-level reflection as available in Java could be treated by augmenting the class objects with additional information, but low-level reflection (e.g. stack/bytecode inspection) and dynamic class loading are obviously at odds with the abstract syntactic approach, just as many other system-level features provided via libraries. Finally, there is another whole array of problems concerned with the semantics multi-threaded memory models. Our viewpoint is that to keep the semantics syntax-based and comprehensible the best solution is to restrict the class of programs to those whose abstract behavior does not rely on the specific memory model. Typechecking techniques for race-free programs like those of [7] look like a promising solution here.

6 Conclusions

The syntax-based approach to semantics has been used to give operational semantics to languages with functional, imperative and/or concurrent features: program equivalance for Scheme-like languages [18, 19, 43]; program equivalence in actor languages [4, 36]; uniform semantics and program equivalence for a family of higher-order imperative languages [49]; interaction equivalence of specification diagrams [44]; and to provide a tool for developing operational semantics and interpreters for programming languages [52]. With the exception of [52], these efforts have not developed executable semantics or automated analyses. To the best of our knowledge the combination of explicit substitutions and reduction contexts to obtain an executable specification of a Felleisen-style semantics is new. It has subsequently been used in a Maude implementation of Specification Diagrams [51].

We have illustrated this syntax-based approach using two applications: PLAN, a packet language for active networks that has been developed and implemented before we started our formalization efforts, and CIAO, a calculus of imperative active objects, that is an example of how language design and prototypeing can go hand-in-hand with the development of a formal semantics. The unique combination of functional/imperative programming and concurrency aspects makes both PLAN and CIAO good candidates to illustrate the use of rewriting logic as a unifying semantic framework. On the conceptual level rewriting logic is general enough to bridge the gap between these different aspects, and on the practical level it comes with an efficient implementation in terms of the Maude rewriting engine, so that in both cases our semantics serves as an executable prototype implementation of the language, and at the same time can be used as a basis for various forms of analysis as well as mathematical or formal reasoning about programs, or more generally about the semantics itself.

It is noteworthy that the syntax-based approach to semantics is not the only approach that is well-supported by rewriting logic and Maude. In fact, several aproaches are discussed in [40], and have been applied to other programming languages including Java [1] and [2]. Both a direct formalization of the Java virtual machine and a continuation-based semantics for a large sublanguage of Java have been developed in these references.

Methodologically, both of our applications suggest a two-step approach to programming language semantics. The first step is to isolate a well-understood and mathematically reasonably elegant core calculus, which in both of our examples is an extension of a call by-value λ-calculus base language. The second step is to define representation mappings from the full language into the core language. In the case of PLAN, the representation mapping into xPLAN is essentially an inclusion, that is PLAN programs can be regarded as a syntactically restricted subset of xPLAN programs [47]. In the case of CIAO, which as we explained is subject of ongoing work, the objective is to further extend the calculus to serve as a core language for multiple real world languages, so that we would be concerned with several different representation mappings, which naturally will be somewhat more complicated. So far we have only given a few examples of the application of such a mapping for Java-like programs, but the mapping remains to be formalized and implemented for a large class of programs. Hence, more work is needed on two fronts: First, for each language of interest (Java, C#, and C++ are obvious candidates) a subset of reasonably well-behaved programs needs to be defined, and second a formal representation mapping for this class of programs needs to be developed. This could again be done in Maude as an executable specification and/or implemented as an external tool.

Another interesting direction that we have experimented with is the use of the executable semantics for *symbolic execution* of programs, that is for the execution of potentially incomplete programs, with holes represented by metavariables. Since such programs can be regarded as the representation of an entire class of conrete programs, symbolic execution could be used like partial evaluation, namely to execute e.g. functions or methods of a program without knowing the concrete arguments. Clearly, symbolic execution can be a useful tool to simplify the analsis and verfication of programs, but there is another interesting application. The abstract computation performed by symbolic execution can then be added itself as a single step rewrite rule to the language semantics, so that each concrete execution which falls ito the class represented by the new rule can be executed in a single step. Due to interation constructs and data/control dependencies there are some limitations to the number of steps that can be covered before the symbolic execution ends, but by studying some example we found that significant speedups can be achieved using this approach, which we simply call *symbolic optimization*, and leave as another interesting area for future work.

Acknowledgments This work has been partially supported by DARPA through Air Force Research Laboratory Contract F30602-97-C-0312, by NSF under Grants CCR-9900326 and CCR-9900334, and by Office of Naval Research Contracts N00012-99-C-0198. and N00014-02-1-0715.

References

1. J. Meseguer A. Farzan and G. Rosu. Formal JVM code analysis in JavaFAN. In *Proceedings of Algebraic Methodology in Software Engineering*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer-Verlag, 2004.
2. J. Meseguer A. Farzan, F. Chen and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proceedings of Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer-Verlag, 2004.
3. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.
4. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
5. K. J. Berkling and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.
6. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
7. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Tampa Bay, FL, USA*, pages 56 – 69. ACM, 2001.
8. K. Bruce. *Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.
9. L. Cardelli. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 286–297, New York, NY, 1995.

10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, January 1999. `http://maude.csl.sri.com`.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on maude. `http://maude.csl.sri.com`, March 2000.
12. O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. In *Second International Workshop on Rule-Based Programming, RULE 2001*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, 2001.
13. *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*. IEEE, January 2000.
14. *DARPA Active Networks Conference and Exposition (DANCE)*. IEEE, May 2002.
15. N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proceedings Kninkl. Nederl. Akademie van Wetenschappen*, volume 75(5), pages 381–392, 1972.
16. G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 251–265. IEEE, January 2000.
17. P. di Blasio, K. Fisher, and C. Talcott. A control-flow analysis for a calculus of concurrent objects. *Transactions in Software Engineering (TSE)*, 2000.
18. M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
19. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
20. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
21. M.J. Parkinson G.M. Bierman and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
22. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*. Elsevier ENTCS, 1998.
23. C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.
24. M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. In *Proceedings of the 1998 Workshop on Internet Programming Languages (IPL'98), Part of IEEE International Conference on Computer Languages (ICCL'98), Chicago, IL, May 1998*, May 1998. `http://www.cis.upenn.edu/~switchware/papers/progplan.ps`.
25. M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September 1998*, pages 86–93. ACM, 1998. `http://www.cis.upenn.edu/~switchware/papers/plan.ps`.
26. M. Hicks and A. D. Keromytis. A secure PLAN. In S. Covaci, editor, *Active Networks, First International Working Conference, IWAN '99, Berlin, Germany, June 30 – July 2, 1999, Proceedings*, volume 1653 of *Lecture Notes in Computer*

Science, pages 307–314. Springer-Verlag, June 1999. `http://www.cis.upenn.edu/~switchware/papers/iwan99.ps`. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

27. M. Hicks, J. T. Moore, and P. Kakkar. Plan programmers guide for PLAN version 3.2. `http://www.cis.upenn.edu/~switchware/PLAN/docs-ocaml/guide.ps`, July 2001.
28. F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.
29. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.
30. L. Cardelli K. B. Bruce and B. C. Pierce. Comparing object encodings. *Information and Computation*, 1999.
31. P. Kakkar. The specification of PLAN. `http://www.cis.upenn.edu/~switchware/PLAN/spec/spec.ps`, 1999.
32. P. Kakkar, C. A. Gunther, and M. Abadi. Reasoning about secrecy for active networks. In *13th IEEE Computer Security Foundations Workshop (CSFW'00), 3 – 5 July 2000, Cambridge, England, Proceedings*, 2000. `http://www.cis.upenn.edu/~switchware/papers/csfw.ps`.
33. P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter. Specifying the PLAN networking programming language. In *HOOTS'99, Higher Order Operational Techniques in Semantics Paris, France, September 30 and October 1, 1999, Proceedings*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.
34. P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 60–69. ACM, 1994.
35. I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.
36. I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.
37. I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer, and C.L. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman*, Lecture Notes in Logic, pages 299–344. Association of Symbolic Logic, 2002.
38. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
39. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
40. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.
41. J. T. Moore, M. Hicks, and S. M. Nettles. Chunks in PLAN: Language support for programs as packets. Technical report, Department of Computer and Infor-

mation Science, University of Pennsylvania, April 1999. `http://www.cis.upenn.edu/~switchware/papers/planchunks.ps`.

42. B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
43. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358, 1993.
44. S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 2002. To appear.
45. M.-O. Stehr. CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ-, σ- and π-calculi. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications Kanazawa City Cultural Hall, Kanzawa Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71 – 92. Elsevier, 2000.
46. M.-O. Stehr. Programming, specification, and interactive theorem proving: Towards a unified language based on equational logic, rewriting logic and type theory. Ph.D. thesis, forthcoming, 2002.
47. M.-O. Stehr and C. L. Talcott. PLAN in Maude: Specifying an active network programming language. In *Fourth International Workshop on Rewriting Logic and its Applications (WRLA'2002), Pisa, Italy, September 19 - 21, 2000*, volume 71 of *Electronic Notes in Theoretical Computer Science*, 2002.
48. M.-O. Stehr and C. L. Talcott. Termination of active network programs. Manuscript, SRI International and University of Hamburg, `http://formal.cs.uiuc.edu/stehr/plan/docs/termination.ps`, February 2002.
49. C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1996. `http://www-formal.stanford.edu/MT/96hoots.ps.Z`.
50. D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
51. P. Thati, C. Talcott, and G. Agha. Techniques for executing and reasoning about specification diagrams. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 521–536, 2004.
52. Y. Xiao, A. Sabry, and Z. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.