Chair of Software Engineering

An Executable Structural Operational Semantics for SCOOP

Master Thesis

Mischael Schill
ETH Zurich
mschill@student.ethz.ch

October 3, 2011 - April 3, 2012

Supervised by:
Benjamin Morandi
Prof. Bertrand Meyer

inf | Informatik Computer Science

Abstract

We report on an implementation of the SCOOP (Simple Concurrent Object Oriented Programming) structural operational semantics in Maude. SCOOP is a programming model that simplifies the writing of concurrent programs. It ensures freedom of race conditions and hides low level abstractions like threads, locks, and memory barriers from the developer. Recently, the team around SCOOP at ETH published a comprehensive structural operational semantics. The next step was to make the structural operational semantics executable to speed up the analysis of programs and the programming model. We chose Maude as the programming language used for implementing the executable operational semantics because it is well-suited for this task and has an active community. The result is a working executable operational semantics which lead to several improvements to the initial semantics.

Acknowledgments

First I'd like to thank my supervisor Benjamin Morandi for all the time and effort he invested to answer my questions and help me if I'm stuck on a problem. He quickly responded when I had a problem and provided useful insight. I'd also like to thank my mentor Prof. Bertrand Meyer for everything I learned from him and all of the contributors to SCOOP for laying the foundation upon which I was able to build this work.

Furthermore I'd like to thank my friend and future wife Simona Wey for her constant support and encouragement.

Contents

Chapter 1

Introduction

Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer defined a comprehensive structural operational semantics of the SCOOP programming model [5]. The next step is to create an interpreter which strictly follows the semantics. This interpreter is useful to develop and test the semantics. The interpreter can be used as a reference implementation for current and future developments and as a research basis new functionality.

The Maude System [3] is a programming language supporting equational and rewriting logic. This makes it very suitable for the task of implementing structural operational semantics. In fact, there are several published examples [7] of structural operational semantics implemented in Maude. Maude also provides a model checker which can check properties formulated in Linear Temporal Logic.

This report presents an implementation of the semantics in Maude. The report is structured as follows. Chapter 2 presents the technique used to translate the semantics into Maude and some details about the translation. Chapter 3 is an example analysis of a program. In chapter 4, the report states the most important contributions to the semantics. Chapters 5 and 6 are the User's and Developer's Guides, the former contains the intermediate representation used in the executable structural operational semantics. Chapter 7 lists the related work and Chapter 8 concludes the report with what can be done in the future.

1.1 Results

A complete interpreter for the SCOOP programming model according to the defined structural operational semantics [5] with an emphasis on correctness, readability and extendability. Having executable operational semantics, it should be possible, using the model checking facility of Maude, to answer questions about the operational semantics of SCOOP. For example, one question arose from the work on the implementation of SCOOP in Eiffel: If we defer the locking of the processors from the beginning of the feature application to the first separate call, do we loose any of the properties established with SCOOP? This could be generalized to questions concerning the impact of any change to the operational semantics of SCOOP.

This work can possibly lead to a publication with the title: "Using an executable operational semantics to finalize the design of a concurrent programming model".

1.2 Short introduction to Maude

1.2.1 Sorts

Maude is based around operators and the sorts theses operators belong to. A sort may have several subsorts, and be the subsort of several sorts. However, a sort may not be its own direct or indirect subsort. Operators belonging to a subsort always belong to all the parent sorts. Equations are used to rewrite terms into other terms and rewrite laws rewrite the state of the program into another state.

```
sort Nat NatSet .
subsort Nat < NatSet .
```

These two lines of Maude code specify the sorts `Nat` for natural numbers and `NatSet` for a set of natural numbers. The second line then declares that natural numbers are also sets of natural numbers (singleton sets).

1.2.2 Operators

Constructors

Constructors are special operators with the `ctor` property. These operators are the basic building blocks because they can not be expanded further. The zero and the successor in the definition of natural numbers are an example of constructors, as is the empty set and the set concatenation.

```
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op empty : -> NatSet [ctor] .
op _,_ : NatSet NatSet -> NatSet [ctor assoc comm id: empty] .
```

The concatenation operator `_,_` is special. The properties `assoc` and `comm` specify that the operator is associative and commutative. The `id: empty` property lists `empty` as the identity for this operator. It also isn't specified in the default prefix notation used by also all other programming language, but in a mixfix notation. The underline symbols the name of the operator specify at which points the first and the second argument are placed.

Regular operators

Other operators are used for abstraction. They usually expand, through several steps of rewriting by equations, into operators that are constructors.

```
op 1 : -> Nat .
op _+_ : Nat Nat -> Nat .
```

Ordinary operators, like constructors, do not need to take any arguments. In this case, they are constants.

1.2.3 Variables

Variables are used in equations and rewrite laws. They are usually declared beforehand:

```
var i j : Nat .
var ns : NatSet .
```

It is possible to use an undeclared Variable. In this case, every time it occurs, it has to be suffixed by a colon and the sort:

```
... a:Nat + b:Nat ...
```

1.2.4 Equations

Equations are used to add the relations between the operators. They are written in a functional style.

```
eq 1 = s(0) .
eq 0 + i = i .
eq s(i) + j = s(i + j) .
eq ns, ns = ns .
```

The first equation gives the constant `1` meaning. The second and third equations recursively calculate the sum of two natural numbers. The last equation make sets of natural numbers idempotent. Pattern matching is very important in Maude. As we can see in the definition of the sum operator, whole terms and not only single operators can stand on the left hand side of the equation. Equation can also have properties, the most important one being `owise`, which lets Maude only use the equation as a last resort if no other equation can be applied.

1.2.5 Rewrite laws

Maude also has the notion of rewrite laws. Rewrite laws are like equations with a few differences: They do not support the `owise` property, because they are inherently non-deterministic. The number of rewrite steps using rewrite laws is controllable by the user, which makes them ideal to represent inference laws. They may also carry a name to identify them.

```
var ns1 ns2 : NatSet .
crl [sum] : ns1, ns2 => (ns1 + ns2)
  if ns1 :: Nat /\ ns2 :: Nat .
```

In this example, we say that two natural numbers in a set can be replaced with the sum of these two. Consider the following expression:

```
1, 2, 3
```

What would the output be? It depends on what Maude chooses to do. If it first adds 1 and 2, the result will be 3. Because before the next step (application of the rewrite rule) is taken, the set is reduced to a singleton containing 3. A singleton can't be rewritten, so the execution stops. However, if it adds 3 and either 1 or 2, the remaining set will be 6.

1.2.6 Conditions and the pattern matching operator

Rewrite laws (`rl`) and equations (`eq`) may have a condition. In this case, they are declared using `crl` and `ceq`. After the right hand side, before the properties, an if clause is inserted (see 1.2.5 for an example). It may contain several expressions, separated by `/\`, and all of them need to be true before Maude applies the rewrite law or equation. It is possible to use the pattern matching operator (`:=`) in these conditions. For example:

```
op nil : -> List [ctor] .
op _ _ : List List -> List [ctor assoc id: empty] .
op _ has _ : List List -> Boolean .
var aList anotherList head tail : List .
ceq aList has anotherList = true
if head anotherList tail := aList .
eq aList has anotherList = false [owise] .
```

This implementation of the `_ has _` operator uses pattern matching in the if clause to check for the occurrence of one list inside of another. The `owise` property is used to only return false if Maude can not apply the conditional equation.

1.2.7 Modules

It is possible to structure a Maude program using modules. Functional modules (`fmod`) may not contain any rewrite rules and may not import any system modules. System modules (`mod`) do not have this restriction. Theories and Views are used for generic programming.

```
fmod NAT is
  sort Nat .
  ...
endfm
fmod NAT-SET is
  including NAT .
  sort NatSet .
  subsort Nat < NatSet .
```

```
  ...
endfm
mod SETCALC is
  including NAT-SET .
  crl [sum] : ...
endm
```

1.2.8 Views

Maude modules may take parameters. Parameters may affect sorts and operators inside such generic modules. These parameters are called views and need to be defined before they are used. We could, for example, user Maude's built-in `SET` module instead of writing our own set. To be able to use our `Nat` sort in a set, we need to create a view. The view is of the theory `TRIV`. This theory is trivial, as the name suggests. It only provides replacement of a single sort, and nothing else. When Maude loads a module that takes a parameter of this theory, it replaces the occurrences of the sort *PARAMETERNAME*`$Elt` with the sort defined in the view. The instantiated module also includes the module specified in the view. It is therefore not possible to include a module that is based on the including module. The generic module may also create generic sorts, they are suffixed with the view name, for example `Set{Nat}`, where `Set` is the name of the sort and `Nat` is the parameter name.

Including a generic module is similar to ordinary module, the parameters are placed in braces after the name of the module. The name of views are usually written in `CamelCase` and reflect the modules involved. For our natural numbers, the view would be the following:

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

We can now write `including SET{Nat} .` This causes Maude to include the Module and replace every occurrence of the sort `X$Elt` by `Nat`. We can use this to make the `NAT-SET` module shorter:

```
fmod NAT-SET is
  including SET{Nat} .
  sort NatSet .
  subsort Set{Nat} < NatSet .
endfm
```

How theories and generic modules are created is beyond the scope of this short introduction.

Chapter 2

Implementation

This chapter describes the implementation of the semantics in Maude. Maude details not described in the short Maude introduction are introduced along with the implementation. Further details on Maude are available in the Maude manual [2].

2.1 Basic support

Maude is powerful, but it also has several limitations that got in the way of implementing the semantics. This section lists the major ones along with solutions to overcome them. These solution provide the basic support for the implementation.

2.1.1 Quantifiers

The semantics often uses quantifiers, which would have been be easy to implement if Maude would support higher order operators. However, Maude does not support passing an operator as an argument. It has a meta-level, which allows manipulation of the program during run-time. This would allow for a work-around, but at great cost of performance. In the future, we want to use Maude to search the state-space of a program, so we can't afford to lose performance.

Solution Many of the quantifiers are expressible through set operations. For example, the `lock_rqs` function of the abstract data type **REGIONS** contains this expression in its require clause:

$$\forall x \in \bar{l} : k.procs.has(x)$$

This can also be expressed using set operators :

$$\bar{l} \subset k.procs$$

However, not all quantifier can be transformed into first order operations. Sometimes, additional operators are needed, either to implement preconditions directly or to provide (filtered) sets.

2.1.2 Identifiers

Maude provides no mechanism for identifiers in the semantics. They are needed to discern between processors, between references, between channels, and between other constructs.

Solution Two mechanisms provide identifiers. A counter inside the state (introduced later in this chapter) provides identifiers for processors and objects. A reference always has the same identifier as the referenced object when the object is added.

2.1.3 Collections

Maude provides collection data types like maps, set, lists and so on. Unfortunately, using one type of collection for more than one sort is problematic. The following example shows this:

```
fmod SORTS is
    *** define some sorts
    sorts A B C D .
    *** make a hierarchy
    subsorts D < B C < A .
endfm
view A from TRIV to SORTS is
    sort Elt to A .
endv
view B from TRIV to SORTS is
    sort Elt to B .
endv
view C from TRIV to SORTS is
    sort Elt to C .
endv
view D from TRIV to SORTS is
    sort Elt to D .
endv
fmod COLLECTION-EXAMPLE is
    protecting LIST{A} + LIST{B}
        + LIST{C} + LIST{D} .
    *** apply the hierarchy also
    *** to the lists
    subsorts List{D} < List{B} List{C}
        < List{A} .
    *** some items for the collections
    op a : -> A [ctor] .
    op b : -> B [ctor] .
    op c : -> C [ctor] .
    *** a list
    op list : -> List{A} .
    eq list = a b c .
```

```
endfm
rew (list) .
```

This example defines a few sorts and lists of all these sorts, not just a list of sort `A`, but also a list that holds only B's and one that only holds `C`'s. A `D` can of course fit into all of these lists. The example then makes sure that the list sorts reflect the same hierarchy as the sorts they contain: a list of `B`'s also qualifies as a list of `A`'s. At last, it tests the list sorts by providing some simple constructors for three items and a list with those. The constructed list should qualify as a list of `A`'s.

No common ancestor When trying out this code, the programmer is greeted with a page full of advisories, which are Maudes version of a warning. Most of them are of the following form:

```
operator nil has been imported from both
"prelude.maude", line 996 (fmod LIST) and
"prelude.maude", line 996 (fmod LIST)
with no common ancestor.
```

This happens because nil is defined in the LIST module, which is imported four times, once for each sort. This can be solved by defining all list operators in a generic way in another module and then let all the specific list modules import it.

Preregularity The programmer also gets some preregularity warnings. Preregularity is achieved when every term in a module has an unique least sort it can be assigned to. If in the example, there wouldn't be a list of `D`'s, then the term `d d` is not preregular. The term qualifies as a list of `A`'s, a list of `B`'s, and as a list of `C`'s. Of all this list sorts, there is not a unique least sort. Because the example has a list of `D`'s, everything is preregular. The problem is, that Maude generates the warnings when loading the list modules and therefore are printed out even though the programmer made sure the list sorts are preregular.

All list elements share a common sort Because the default `LIST` module defines the element sort as a subsort of the list, suddenly many more problems arise. Because `Int` and `Bool` both define some operators, but with different properties, new error messages arise. This means either no lists of integers or no lists of boolean values. This can be avoided if a single element is not automatically interpreted as a list or a set.

Solution To solve the issues above, the implementation uses new collection modules that avoid these issues.

2.1.4 Target language syntax

Maude provides a mixfix notation for operators. When implementing the SCOOP syntax using this mixfix operators, several problems occured. For one, many constructs

(like the assignment operator `:=`) clash with Maude constructs. This results in parsing errors.

Furthermore, it is hard to implement optional parts because Maude doesn't directly support it. It would be possible to simply have one operator for each combination of optional elements, but this leads to an exponential explosion of operators.

Finally, the SCOOP syntax uses commas, semicolons and white spaces in lists. Supporting all of these in Maude is complicated.

Solution The executable operational semantics uses an intermediate representation that is more friendly to Maude and minimizes the amount of code. The syntax of this intermediate representation is documented in the user's guide.

2.2 Program syntax

Maude has a powerful mixfix notation. As explained in section 2.1, Maude is not powerful enough to interpret SCOOP syntax directly. For this reason, the implementation uses an intermediate representation based on Maude's mixfix notation.

The executable semantics can not parse SCOOP code directly. The input has to be in an intermediate representation which is also used internally. The parser expects white space after every token except for parentheses and curly braces.

In this section, blocks in parentheses that are followed by a question mark, a star or a plus symbol are not part of the syntax but instead used as in regular expressions. A pipe symbol (`|`) inside a bracketed block indicates a choice in the syntax, these parentheses are also not part of the syntax.

2.2.1 Identifiers

Class, feature, and variable names are always preceded with an apostrophe (`'`). The same is true for the special variables `Result` and `Current`.

2.2.2 Maude configuration

Every program should start with the following instructions. They instruct Maude to conceal classes and features.

```
set print conceal on .
print conceal ( _ _  create _ _ invariant _ end ) .
print conceal ( procedure { _ } _ ( _ ) require _ local _ _ _ ensures _ end ) .
print conceal ( function { _ } _ ( _ ) :
  _ require _ local _ _ _ ensures _ end ) .
```

2.2.3 Rewrite

Maude rewrites the input step by step. Therefore, the entire program has to be an argument to the rewrite command:

```
rew [1] (
<PROGRAM>
) .
```

This instructs Maude to rewrite the program and start with one step. This step will transform the input program into the initial configuration.

2.2.4 Program

A program consists of a list of classes and the root procedure. The term `import default` inserts the default classes (`BOOLEAN` and `INTEGER`) into the program. After every class (but not after `import default`), a semicolon needs to be appended:

```
<PROGRAM> ::=
(
import default
(<CLASS> ;)+
) {<CLASSNAME>} . <FEATURENAME>
```

2.2.5 Class

A class consists of its name, the set of constructor names, the list of features and the invariant:

```
<CLASS> ::=
class <CLASSNAME>
  create
    {<FEATURENAME>} (U {<FEATURENAME}})*
  (
  (<FEATURE> ;)+
  )
  invariant
    <EXPRESSION>
end ;
```

If there is no invariant for the class, the expression should simply be `True`. The set of constructor names may also be `empty`, but this is only useful for classes that have a literal representation like `INTEGER` and `BOOLEAN`.

2.2.6 Features

There is no fixed ordering for features. Different from the SCOOP syntax, features can not be organized in blocks and are preceded by the kind of feature instead of

the `feature` keyword. For this version of the interpreter, the class name in the curly braces is currently only relevant for indicating whether the feature is exported or not. If not needed, the expressions for the `require` and `ensures` clauses should be `True`.

```
<FEATURE> ::= (<ATTRIBUTE> | <PROCEDURE> | <FUNCTION>)
```

Attributes

```
<ATTRIBUTE> ::=
attribute {<CLASSNAME>} <FEATURENAME> : <TYPE>
```

Procedures

```
<PROCEDURE> ::=
procedure {<CLASSNAME>} <FEATURENAME>
    ( (nil | (<ENTITYDECLARATION> ;)+) )
  require
    <EXPRESSION>
  local
    (nil | ( (<ENTITYDECLARATION> ;)+ ))
  (do | once)
    (nil | ( (<INSTRUCTION> ;)+ ))
  ensures
    <EXPRESSION>
  end
```

Functions

```
<FUNCTION> ::=
function {<CLASSNAME>} <FEATURENAME>
    ( (nil | (<ENTITYDECLARATION> ;)+) ) : <TYPE>
  require
    <EXPRESSION>
  local
    (nil | ( (<ENTITYDECLARATION> ;)+ ))
  (do | once)
    (nil | ( (<INSTRUCTION> ;)+ ))
  ensures
    <EXPRESSION>
  end
```

2.2.7 Types

```
<TYPE> ::= [<DETACHABLETAG>, <PROCESSORTAG>, <CLASSNAME>]
```

A type consists of a triple containing:

1. The detachable tag: `!` (attached) or `?` (detachable)
2. The processor tag: `T` (separate) or `*` (non-separate)
3. The name of the class.

An attached `INTEGER` type would look like this:

```
[!,*,'INTEGER]
```

A detachable, separate `CHANNEL` would be:

```
[?,T,'CHANNEL]
```

2.2.8 Entity declarations

```
<ENTITYDECLARATION> ::= <VARIABLENAME> : <TYPE>
```

Entity declaration are used for the declaration of arguments and local variables.

2.2.9 Instructions

```
<INSTRUCTION> ::= (<ASSIGNMENT> | <COMMAND> | <IF> | <LOOP>)
```

Assignments

```
<ASSIGNMENT> ::=
assign(<VARIABLENAME>, <EXPRESSION>)
```

Commands (procedure calls)

```
<COMMAND> ::=
command(<EXPRESSION>)
```

Ifs

```
<IF> ::=
if <EXPRESSION> then
  (nil | ( (<INSTRUCTION> ;)+ ))
else
  (nil | ( (<INSTRUCTION> ;)+ ))
end
```

Loops

```
<LOOP> ::=
until <EXPRESSION> loop
  (nil | ( (<INSTRUCTION> ;)+ ))
end
```

2.2.10 Expressions

```
<EXPRESSION> ::=
  (<LITERAL> | <QUERY> | <VARIABLENAME> | <FEATURENAME>)
```

Literal

```
<LITERAL> ::=
  (<INTEGER> | <BOOLEAN>)
<INTEGER> ::=
  (-)?(<DIGIT>)+
<DIGIT> ::= (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
<BOOLEAN> ::= (True | False)
```

Query

```
<QUERY> ::=
  <EXPRESSION> . <FEATURENAME> ( (nil | (<EXPRESSION> ;)+) )
```

2.3 Translating the state

The semantics uses abstract data types (ADT) to model the state. These abstract data types can be translated in a few steps, explained in this section. The ADT **OBJ** for objects is used as an example to show the process. These are all the ADT's in the semantics:

ID Identifiers, replaced by natural numbers (NAT) in the implementation

REF References

OBJ Objects, mapping of attributes to references

PROC Processors

VALUE References and objects (**REF** $\cup$ **PROC**)

REGIONS The regions, locking information and the relation between objects and processors

HEAP The heap, mapping of references to objects

ENV Environments, mapping of local variable names to references

STORE The store, mapping of processors to stacks of environments

STATE The state of the program

2.3.1 Names of sorts, modules and operators

Maude uses modules to structure a program. To avoid confusion between module and sort in case a module encapsulates a sort and associated operators, module names are written in `CAPITALS`, and sort names are written in `CamelCase`. In Maude, the underline (_) symbol has special meaning. Because of this, all operators are named in `camelCase` instead.

2.3.2 Sorts

Each ADT has its own sort. Sometimes, additional sorts are needed due to of shared functionality. The sorts is placed in a separate module. This is needed when an ADT contains an operator that takes a list of elements of the same sort as the ADT. Because the list it includes also includes the module where the sort is declared, this results in a circular dependency. It makes sense to put the subsort declaration next to the sort declaration in the same module.

The following defines the sort `Obj` and makes it a subsort of `Identifiable`. This sort is shared by all sorts that have the `.id` query to retrieve the identifier. In order to have access to the `Identifiable` sort, the module `IDENTIFIABLE` is included. This module also provides the necessary operator declaration for `.id`.

```
fmod OBJ-SORTS is
  including IDENTIFIABLE .
  sort Obj .
  subsort Obj < Identifiable .
endfm
```

2.3.3 Views

Before `Obj` can be used as a parameter to a module, a view is needed to tell Maude what to do when it encounters `Obj` as a parameter to a module that allows a parameter of the theory `TRIV`. The implementation only uses module parameters for collections, and all these parameters are from this theory. The declaration of a view that replaces the generic elements of a module by objects is:

```
view Obj from TRIV to OBJ-SORTS is
  sort Elt to Obj .
endv
```

When a module includes the module `SETObj`, The set sort provided by this would then be `SetObj` to differentiate between sorts imported from the same module with different parameters.

2.3.4 Includes

Each ADT has a second module for the definition of the operators and equations. It starts by including all the modules it relies upon.

```
fmod OBJ is
  including OBJ-SORTS .
  including SET{Attribute} .
  including VALUE .
  including REF .
  including MAP{Name, Value} .
  including PROGRAM .
  including SET{Ref} .
  including SET{Value} .
  including SET{Obj} .
```

2.3.5 Constructors

The next step is to define at least one constructor for each ADT.

```
  op obj : Nat Class Map{Name, Value} -> Obj [ctor] .
  op null : -> Obj [ctor] .
```

The first constructor is an operator in prefix notation. It takes three arguments and is of sort `Obj`. The three arguments are the id, the type and the attribute values of the object. The second constructor is a constant. It takes no arguments and is used to denote a non-existing object.

2.3.6 Operators

The module goes on with the regular operators to define the queries and commands of the ADT. The operator precedence of 10 is important to support chains of operators.

```
  op _.id : Obj -> Nat [prec 10] .
  op _.classType : Obj -> Class [prec 10] .
  op _.attVal (_) : Obj Name -> Value [prec 10] .
  op _.setAttVal (_ , _) : Obj Name Value -> Obj [prec 10] .
  op _.refs : Obj -> Set{Ref} [prec 10] .
  op new OBJ.make (_, _) : Class Nat -> Obj [prec 10] .
  op _.copy(_) : Obj Nat -> Obj [prec 10] .
  op _.obj(_) : Literal Nat -> Obj [prec 10] .
  op $initAttributes : Set{Attribute} -> Map{Name, Value} .
  op _.filterRefs : Set{Value} -> Set{Ref} [prec 10] .
```

The definition of the operators is similar to the definitions in the semantics. Some operators have an additional natural number (`Nat`) as an argument; this is the identifier. There are more operators than specified because some of them are needed for

the implementation of others. For instance, Maude neither has quantifiers nor higher order functions. The dollar sign before `initAttributes` is a hint that the operator is only used within the module.

2.3.7 Equations

Maude uses equations to simplify terms, the implementation uses them to implement the axioms of the semantics. An equation may be conditional, which is used by the implementation for the preconditions. Maude has powerful pattern matching, which respects commutative, associative, and idempotent operators. This can make recursion unnecessary. For example, looking through an associative list for a single element can be phrased as a matching a head list, the element and the tail list on the original list. The element may also have some arbitrary condition placed on it. The variables used in equations need to be declared beforehand:

```
  var id id2 : Nat .
  var classType : Class .
  var attVals : Map{Name, Value} .
  var f : Attribute .
  var n : Name .
  var as : Set{Attribute} .
  var i : Int .
  var v : Value .
  var vs : Set{Value} .
```

Maude tries to always simplify equations until it reaches a dead end. Some of the equations for objects are:

```
  eq obj(id, classType, attVals) .id = id .
...
 eq obj(id, classType, attVals) .refs = attVals .values .flat .filterRefs .
 eq obj(id, classType, attVals) .copy(id2) = obj(id2, classType, attVals) .
  eq new OBJ.make (classType, id)
    = obj(id, classType, $initAttributes(classType .attributes)) .
 eq True .obj(id) = obj(id, BOOLEAN, (empty .insert('item --> B(true)))) .
  eq False .obj(id) =
    obj(id, BOOLEAN, (empty .insert('item --> B(false)))) .
  eq (i) .obj(id) = obj(id, INTEGER, (empty .insert('item --> I(i)))) .
  eq $initAttributes(({f} U as)) =
    $initAttributes(as) .insert(f .name --> void) .
  eq $initAttributes(empty) = empty .
...
endfm
```

With the equations defined, the translation of an ADT into Maude code is finished.

2.4 State details

As seen in the previous section, it is impossible to convert the semantics without modification into Maude code. For every ADT, a constructor needs to be derived and all quantifiers in preconditions have to be converted into first order expressions. There are also cases where the semantics uses an ellipsis to indicate a simple recursion. To achieve this in Maude, it is sometimes necessary to introduce additional operators. This section explains all the derived constructors, the additional operators, and the differences to the semantics.

2.4.1 Value

The `Value` sort is the common ancestor for references and processors. It is used wherever **REF** $\cup$ **PROC** is specified in the semantics.

2.4.2 Reference

Constructor

```
op ref : Nat -> Ref [ctor] .
op void : -> Void [ctor] .
```

The main reference constructor has one argument, which is the identifier. The second constructor is used when dealing with void references.

Differences

The creation operator takes a natural number for the identifier as an additional argument.

2.4.3 Object

Constructor

```
op obj : Nat Class Map{Name, Value} -> Obj [ctor] .
op null : -> Obj [ctor] .
```

The main object constructor has three arguments:

1. The identifier of the object
2. The class this object belongs to
3. A map from attribute names to values

There is also a `null` constructor that is used when no specific object should be referenced.

Differences

The creation operator `make` takes a natural number as argument to set the identifier. For readability, the map from attributes to values is implemented as a map from names to values. Otherwise, either the long attribute definition would make reading hard or the collapsed attributes would make it impossible to distinguish between the attributes in the output

2.4.4 Heap

Constructor

```
op heap : Map{Ref, Obj} Map{ProcNamePair, Ref} Obj
  -> Heap [ctor format(ni d)] .
```

The heap constructor has three arguments:

1. A map from references to objects
2. A map from processors and names to references to save the once results
3. The object added last

Differences

The `.addObj` operator takes an identifier as an additional argument to set the identifier of the reference created by the operator. It uses the new operator `.objIdUnique` to make sure the new object does not have an identifier that is already in use.

Tracking of separate once function results

In SCOOP, separate once function results are tracked globally. To achieve this, all these results are tracked on the root processor. This affects the operators `.isFresh` and `.onceResult`. There, the processor that tracks the result is determined as follows (The processor `p` and feature `f` are the arguments to these operators):

```
q := if f .resultType .processorTag =/= • then
  new PROC.make(0) else p fi
```

2.4.5 Processor

Constructor

```
op proc : Nat -> Proc [ctor] .
op noProc : -> Proc [ctor] .
```

The `proc` constructor takes the processor id as its only argument. The `noProc` constructor is used to denote no specific processor.

Differences

The creation operator takes a natural number for the identifier as an additional argument.

2.4.6 Regions

Constructor

```
op regions : Map{Proc, ObjSet} Proc Map{Proc, Bool} Map{Proc, Bool}
  Map{Proc, ProcSetList} Map{Proc, Proc} Map{Proc, ProcSetList}
  Map{Proc, ProcSetList} Map{Proc, Bool}
  -> Regions [ctor format(ni d)] .
```

The regions constructor has multiple arguments:

1. A map from each processor to all the objects it handles
2. The processor added last
3. A map indicating which request queues are locked
4. The same for the call stack (always true)
5. A map from each processor to the stack of lock sets with the obtained request queue locks
6. The same for the obtained call stack lock
7. The same for the retrieved request queue locks
8. The same for the retrieved call stack locks
9. A map indicating the passed lock state of each processor

Differences

Because delegated postcondition evaluation has been determined to be useless, there is no operator `.delegateObtainedRqLocks`.

Additional operators

The `.addObj` operator, like the operator with the same name in the heap, uses the new operator `.objIdUnique` to make sure that the new object does not have an id that is already in use by any processor.

To replace several quantifiers, the new operators `.areRqsOpen`, `.areRqsLocked`, and `.allRqLocks` are introduced. The first checks that a set of processors has their request queues unlocked, the second checks that their request queues are locked, and the last returns all locks in the system.

A new operator named `.flatValues` returns a set of all the processors that are somewhere in the stack of processor sets. The `.removeObjs` operator removes a set of objects from the regions.

2.4.7 Environment

Constructor

```
op env : Map{Name, Value} Feature -> Env [ctor] .
```

The constructor for an environment takes a map from attribute names to values and a single feature as arguments. The feature argument is the feature to which this environment belongs. This enables the state to determine entity declarations from entity names.

Differences

The creation operator of an environment takes an additional argument with the feature associated to this environment. This feature can then be queried using the new `.feature` operator.

Additional operators

The `.refs` operator returns all the non-void references in a environment. The new `.feature` operator is used to get the feature running on this environment.

2.4.8 Store

Constructor

```
op store : Map{Proc, EnvList} -> Store [ctor format(ni d)] .
```

The constructor takes a map from each processor to a stack of environments.

Additional operators

The `.refs` operator returns all non-void references stored in all environments.

2.4.9 State

Constructor

```
op state : Regions Heap Store Ref Nat
  -> State [ctor format(ni d)] .
```

The state constructor takes the regions, the heap, and the store. In addition, it has arguments for the last imported reference and the identifier counter.

Identifier management

The three operators `.idc`, `.incId`, and `.addId` query and control the identifier counter. The `.idc` operator returns the next identifier to assign. Afterwards, the `.incId` is used to increment the counter. The `.addId` command is used to add more than one to the identifier counter and can be used if several consecutive identifiers are assigned at the same time.

Because the identifier counter inside the state needs to increase after an object or processor is created, the state now has operators called `.addNewProc` and `.addNewObj` which combine the `.newProc` and `.addProc` resp. `.newObj` and `.addObj` operators. Some of the inference rules needed to be modified slightly to support this.

Processors and objects New processors and objects need an identifier upon creation. Therefore the `.newProc` and `.newObj` operators take an additional argument with the identifier. To automate the assignment of identifiers to processors, the new operator `.addNewProc` is used instead of first calling `.newProc` and then `.addProc`. This operator is a combination of these two operators. It assigns the next identifier to the processor and adds it to the state. The operator `.addNewObj` works the same for objects by combining `.newObj` and `.addObj`.

Additional operators

The `.handlers` operator works like the `.handler` operator but takes a list of references and returns a set of processors instead of a single reference and a single processor. With `.currentFeature`, it is possible to get the feature that is currently executing on a processor. To filter a list of processors to only those that have their locks passed, the `.filterPassedLocks` operator can be used. The `.filterNonExpanded` operators filters a list of references so that only references to non expanded objects remain.

The deep import operator uses several helper operators. `.setLastImportedRef` is used by the deep import operation to set the last imported reference. The new `$filterNonVoidAttributes` operator filters a list of feature names so that only those attributes remain whose value is a non-void reference. The new `$replaceAtts` operator is used to replace attribute values according to a supplied map.

The new `.pushEnv` operator also uses additional internal operators. The new `$pushEnvHelper` operator processes the argument list and the `$recusiveEnvUpdate` sets up the environment.

The `.setVal` operator needs the `$copyObj` helper operator to copy an object if it is expanded and return it without modification otherwise.

Garbage collection

Garbage collection is implemented using three operators: `.gc`, the main operator, `.mark`, an operator that creates a set of reachable objects from a root set, and `.sweep`, which removes a set of objects from the heap and the regions. The `.gc` operator takes a set of objects. This set must contain all the references that are arguments to operations in any action queue.

2.5 Translating configurations

In the semantics, the program is hidden and the typing environment Γ is not part of the configuration. In Maude, we can not have a static, hidden part. To access the program or the typing environment, this information has to be part of the term that is rewritten. The solution is the following operator definition for the configuration:

```
op _ |- _ , _ , _ : Program ActionQueueList Nat State
  -> Configuration [ctor prec 121] .
```

The first argument is the program to execute. The letter Γ is used for this. The program is static and also contains the typing environment. By adding it into the configuration, every rewrite law has access to it. The second argument is a list of all the action queues. The third argument, a natural number, counts the number of steps the program has taken. It is used as the first argument for fresh channels. This way it is easy to determine in which step a channel has been created.

2.5.1 Action Queue List

```
op nil : -> ActionQueueList [ctor] .
op _ | _ : ActionQueueList ActionQueueList
  -> ActionQueueList [ctor assoc comm id: nil prec 122] .
```

The action queue list is commutative and separated by pipes (|). `nil` is used as the constructor for an empty action queue list.

2.5.2 Action Queue

```
op {_} _ :: _ : Nat Proc List{Statement}
  -> ActionQueue [ctor prec 121] .
```

The action queue takes a natural number, the processor and a list of statements as its parameters. Statements can be both instructions (supplied by the user) and operations (used only internally). The natural number is the priority of a processor. When executing a program, Maude always takes the first match when having to make a non-deterministic choice. It orders all the commutative operators and then searches for possible matches from left to right. So by adding an additional argument, we can influence the ordering and the choices Maude makes. This enables us to create and

test scheduling algorithms. When searching the state space, Maude ignores these numbers.

2.5.3 From the input to the initial configuration

A rewrite law may not rewrite one sort into another. So the program an the name of the initial feature has to be a valid configuration too.

```
op ( _ { _ } . _ ) : Program Name Name
  -> Configuration [ctor format (d n s s s s n)] .
```

An additional rewrite law takes care to create the initial configuration from there. It uses the new commands `.addNewProc` and `.addNewObj` for automatic identifier assignment.

```
crl [init] : Γ { cn } . fn =>
  Γ |- {0} p ::
    call(r, fn, nil, nil);
    issue(q, unlock ;);
    popObtainedRqLocks ;
    | {0} q :: nil, 1, σ
  if
    c := Γ .classByName (cn) /\
    σx := new STATE.make /\
    σy := σx .addNewProc /\
    p := σy .lastAddedProc /\
    σz := σy .addNewProc /\
    q := σz .lastAddedProc /\
    σw := σz .addNewObj(c, q) /\
    r := σw .ref (σw .lastAddedObj) /\
    σ := σw .lockRqs(p, {q}) .
```

2.6 Translating operations

Operations are always constructors. All operations except for `provides` are implemented as prefix operators.

2.6.1 Creation of multiple operations

The semantics sometimes uses `eval`, `wait` and `issue` operations multiple times in a row. To accomplish this, the operators `neval`, `nwait` and `nissue` have been introduced. They immediately unfold to a list of statements. Their arguments are similar to their single operation counterpart, but they take lists or sets of arguments instead.

2.6.2 Reference retrieval

The garbage collector needs a set of all references in all the action queues. The `scan` operator scans every statement and produces a set of references. To support this, equations are needed for every operation that contains a reference or a list of statements.

2.7 Channels

The `[_/_]` operator is used to replace the `channel().data` placeholders with the actual reference. For this to work, every operation and instruction has equations that handle the substitution. There is a fallback equation that simply ignores the statement; this way, only the cases where the substitutions affects a statement need to be covered with equations.

Channels have three natural numbers as identifiers. The first one is the step where the channel was created, which is the natural number located between the list of action queues and the state. The second one identifies the `fresh` function in a rewrite law and the last one is used by `nfresh` to create multiple channels at once.

2.8 Translating inference rules

Inference rules are translated into rewrite laws.

2.8.1 Translating the fresh routine application rule

The most complex inference rule is the one for non-once or fresh once routines. It takes a result channel, the target, a feature, a list of arguments, the calling processor and the passed locks and applies it on the current processor. This makes it a good example on how to translate complex inference rules.

The first step is to create a skeleton for the inference rule:

```
crl [applicationOpNonOnceOrFresh] :
  Γ |- {i} p :: apply(a, r0, f, rs, q, l) ; sp, ic, σ =>
    Γ |- {i} p ::
    ...
    sp, ic + 1, σ''
  if
    ...
    .
```

The next step is to add the right hand side of the transition:

```
    checkPreAndLockRqs(q, gmissingRQLocks:Set{Proc}, f) ;
    provided f :: Function and f .isOnce then
```

```
    replaceOnceResult(f, f .body)
  else
    f .body
  end ;
  checkPostAndUnlockRqs(gmissingRQLocks:Set{Proc}, f) ;
  provided σ .refObj(r0).classType .invExists
      and f .isExported then
    eval (ainv, σ .refObj(r0).classType .inv) ;
    wait(ainv) ;
  else
    nop ;
  end ;
  provided f :: Function then
    read('Result, a') ;
    return(a, a' .data, q) ;
  else
    return(a, q) ;
  end ;
```

The main difference to the semantics is the use of the `replaceOnceResult` operator to manipulate the feature body of a once routine to update the once result in the state. Another difference is how the class type of the feature is determined. It is not possible to add the class to the constructor of a feature. A class contains its features and if the feature would contain the class it belongs to, this would be a circular relationship. Maude does not have pointers so this is impossible. The class type is determined by looking up the object on which the feature is executed and then get the class type from the object.

The premise of the inference rule is added as a condition on the rewrite law:

```
(f :: Routine and f .isOnce) implies σ .isFresh(p, f) /\
σ .handler(r0) == p /\
σ' :=
  if (f :: Function and f .isOnce) then
    σ .setOnceFuncNotFresh(p, f, void)
  else
    if f :: Procedure and f .isOnce then
      σ .setOnceProcNotFresh(p, f)
    else
      σ
    fi
  fi /\
σ'' := σ' .passLocks(q, p, l).pushEnvWithFeature(p, f, r0, rs) /\
not σ'' .areLocksPassed(p) /\
grequiredLocks:Set{Proc} :=
  {p} U handlersOfAttachedFormals(Γ, σ'', f .formals, rs) /\
grequiredCSLocks:Set{Proc} :=
  {p} U σ'' .filterPassedLocks(grequiredLocks:Set{Proc}) /\
grequiredRQLocks:Set{Proc} :=
  grequiredLocks:Set{Proc} \ grequiredCSLocks:Set{Proc} /\
```

```
    gmissingRQLocks:Set{Proc} :=
      grequiredRQLocks:Set{Proc} \ σ'' .rqLocks(p) /\
    grequiredCSLocks:Set{Proc} ⊂ σ'' .csLocks(p) /\
    ainv := fresh(ic, 1) /\
    a' := fresh(ic, 2)
```

All the quantifiers had to be replaced by equivalent set operations. Apart from this, the condition is the same as the premise of the inference rule.

2.8.2 Parallelism

The parallelism inference rule can be translated into Maude like this:

```
crl [parallelism] : Γ |- aqs1 | aqs' , ic, σx
  => Γ |- aqs2 | aqs', ic', σy .gc(filterRefs(scan((aqs2 | aqs'))))
  if
    not aqs' == nil /\
    Γ |- aqs1, ic, σx => Γ |- aqs2 , ic', σy /\
    not (aqs1 == aqs2 and σx == σy) .
```

There are a few differences. The first and the last condition are needed to ensure that the parallelism rule is not applied indefinitely in one step. Because the first rule makes sure the number of action queues in the focus is getting smaller and smaller, at some point it can not be applied anymore. The last condition makes sure that something changes in the system. The garbage collector is also hooked into this rule.

2.8.3 Inference rules without using typing environment

The typing environment is not yet complete. This means that the `is_controlled` function is not available. The implementation instead checks that the current processor has a lock on the processor on which the object resides. Without explicit processor tags and assuming correctly typed input programs, this is equivalent to the call to `is_controlled`.

2.8.4 Yielding

When executing a program, Maude tries to go forward with the uppermost processor in its list, even though the list is commutative. Without involving the meta-level, which would incur a heavy performance penalty, there is no way to tell Maude which rule should be applied next. By adding a priority to every action queue, Maude sorts the list accordingly and tries to apply rules to the processors with the lowest priority number first.

The initial processor starts with priority 0. Every time a processor spawns another, the new processor gets the same priority. The `yield` operation increases the priority by one. Every time a precondition fails, the processor yields so that others can go forward.

```
rl [yield] :
  Γ |- {i} p :: yield ; sp, ic, σ => Γ |- {i + 1} p :: sp, ic + 1, σ .
```

2.8.5 Internal calls

Some support is needed to implement the `INTEGER` and `BOOLEAN` classes and to provide a means to determine equality between values of these types. These are always prefixed with a dollar-sign to distinguish them from regular calls. They are implemented using rewrite laws. Because they are all very similar, there is no need to document all of them here. The "greater than" operator serves as an example:

```
crl [integerGreater] :
  Γ |- {i} p :: call(a, r0, '$gt, rs, es) ; sp, ic, σ
  => Γ |- {i} p :: result(a, r); wait(a); sp, ic + 1, σ'
  if
    I(a:Int) := σ .refObj(r0) .attVal ( 'item ) /\
    I(b:Int) := σ .refObj(rs .top) .attVal ( 'item ) /\
    σ' := σ .addObj(p,
      obj((σ .idc), BOOLEAN, ('item --> B(a:Int > b:Int)))
      ).incId /\
    r := σ' .ref(σ' .lastAddedObj) .
```

It hooks directly into the call operation. If the name of the called feature is `$gt` and both arguments are integers (ensured by the first two conditions), a new `BOOLEAN` object is created with the value equal to the "greater than" comparison of the two arguments.

Chapter 3

Example

This chapter presents an example run. Because listing the whole source code and all the output would be far too much, only some interesting parts are presented. The state is described in a small text instead of providing the whole listing. The source to both examples is part of the distribution of the executable operational semantics.

A reader and a writer use the same channel. The channel has a size of one integer. The writer writes two integers into the channel and the reader reads both. Because of the limited size of the channel, the writer has to wait until the reader read the first integer before writing the second. This is a simple example to show how two processors access a third with a synchronization condition.

3.1 Initialization

After the initialization, the action queues of the two processors look like this:

```
{0}proc(0) ::
call(ref(2), 'make, nil, nil) ;
issue(proc(1),
  unlock ;
) ;
popObtainedRqLocks ; |
{0}proc(1) :: nil
```

The bootstrap processor will call the `make` feature on the object referenced by `ref(2)`. For this, it needs the lock processor 1. After the call, it lets processor 1 unlock itself and releases the locks.

Looking at the state, we can see the following:

- `ref(2)` points to `obj(2)` which is an instance of `CHANNEL_SIMULATOR`.
- `obj(2)` belongs to processor 1.

- The request queue of processor 1 is locked.
- processor 0 obtained processor 1's request queue lock.

3.2 The first step

After taking one step using `cont 1 .`, the following happened:

```
{0}proc(0) ::
issue(proc(1),
  apply(channel(1, 1, 0), ref(2), procedure 'make (...), nil,
    proc(0), [empty,empty]) ;
) ;
provided false then
  wait(channel(1, 1, 0)) ;
else
  nop ;
end ;
(...)
```

The call operation succeeded and was replaced with an issue operation. This operation lets the processor 1 apply the procedure `make`. Because processor 1 does not need any locks to apply this procedure, the following `wait` operation is skipped. As we can see, the expression in the provided clause is already reduced to `false`.

3.3 Issuing the first feature application

Until now, only processor 0 had something to do. This changes now. The issue operation also involves processor 1.

```
{0}proc(0) ::
provided false then
(...) |
{0}proc(1) ::
apply(channel(1, 1, 0), ref(2), procedure 'make(...), nil,
  proc(0), [empty,empty]) ;
```

The `apply` operator has been moved to processor 1. Now Maude has to decide whether it wants to execute the `provided` operation on processor 0 or the application operation on processor 1.

3.4 The first non-deterministic choice

```
{0}proc(0) ::
nop ;
```

```
issue(proc(1),
(...)
```

It chose to go ahead with processor 0. When used as an interpreter, Maude usually prefers the processors higher up in the action queue list. That is why there is the priority flag. It does not limit the possibilities but nudges Maude in the right direction when it simply executes code.

The next step will be a `nop` operation, which is not interesting. The same is true for the issuing of the unlock operation afterwards. Using `cont 3 .` we take three steps at once.

3.5 Releasing locks

Maude just executed the `popObtainedRqLocks` operation. This changed something in the regions part of the state. Processor 0 has processor 1 no longer in its stack of obtained request queue locks. Processor 1 is still marked as locked, because processor 1 is not done with the feature application:

```
{0}proc(0) :: nil |
{0}proc(1) ::
apply(channel(1, 1, 0), ref(2), procedure 'make(...), nil,
  proc(0), [empty,empty]) ;
unlock ;
```

Processor 0 is done. It will not do anything for the rest of the execution, so we can silently drop it from the listings from now on. We can also see that the unlock operation has not been executed yet. It is set at the end of the action queue of processor 1 because the processor needs to first do the work it has been issued to do before accepting something else.

3.6 First feature application

It is time for processor 1 to wake up and do something.

```
{0}proc(1) ::
checkPreAndLockRqs(proc(0), empty, procedure 'make(...)) ;
provided false then
  (...)
else
  create('channel . 'make(nil)) ;
  create('writer . 'make('channel ;)) ;
  create('reader . 'make('channel ;)) ;
  command('Current . 'simulate('writer ; 'reader ;)) ;
end ;
checkPostAndUnlockRqs(empty, procedure 'make(...)) ;
```

```
provided false then
  eval(channel(7, 1, 0), True) ;
  wait(channel(7, 1, 0)) ;
else
  nop ;
end ;
provided false then
  read('Result, channel(7, 2, 0)) ;
  return(channel(1, 1, 0), channel(7, 2, 0) .data, proc(0)) ;
else
  return(channel(1, 1, 0), proc(0)) ;
end ;
unlock ;
```

The amount of work for processor 1 just exploded. It needs to first check the preconditions of the procedure and, if necessary, lock the request queues. It then has to decide whether it needs to update once results. It does not, because the executed feature is a regular procedure. After executing the body of the procedure, it has to check the postcondition and free all the locks again. There is no class invariant (it is set to `True`), so the processor at least does not have to evaluate an invariant. The feature is a procedure, so there is no result to return.

We can see the four instructions making up the body of this procedure by looking at the first `provided` operation.

The state also changed:

- The stacks of retrieved request queue and call stack locks are bumped up with an empty set.
- The stack of environments for processor 1 in the store now contains an environment with the `Current` variable pointing to `ref(2)`.

3.7 The first instruction

Because there is no precondition and no need for locking, we let Maude run seven steps.

```
{0}proc(1) ::
create('channel . 'make(nil)) ;
create('writer . 'make('channel ;)) ;
create('reader . 'make('channel ;)) ;
command('Current . 'simulate('writer ; 'reader ;)) ;
(...)
```

Processor 1 now has to execute the body of the procedure. The first step is to create a new channel. The entity `channel` is a local variable of type `CHANNEL`. The state did not change much during the last seven steps. Because the procedure applications does not need any additional locks, the stack of retrieved request queue locks for processor 1 was bumped up with an empty set.

```
{0}proc(1) ::
lock({proc(3)}) ;
write('channel, ref(4)) ;
command('channel . 'make(nil)) ;
provided true then
  nop ;
else
  issue(proc(3),
    eval(channel(15, 1, 0), True) ;
    wait(channel(15, 1, 0)) ;
  ) ;
end ;
issue(proc(3),
  unlock ;
) ;
popObtainedRqLocks ;
(...) |
{0}proc(3) ::
nop ;
```

Instead of the creation instruction, we now have several other statements. There is also a new processor involved, processor 3. The state also contains a few new things. There is the new reference 4 that points to object 4, which is owned by processor 3. Processor 3 is still free, but this changes with the lock operation that is executed next. After locking, the new reference is written into the variable `channel`.

3.8 Locked and written

The two operations did not produce anything new in the action queues. But they had an effect on the state: Processor 3's request queue lock is obtained by processor 1 and the `channel` variable in the current environment of processor 1 points to `ref(4)`.

Up next is the constructor of `channel`. This is not interesting because we have already seen a call to a constructor. To skip the creation instructions we let Maude execute until step 174.

3.9 Local command with separate arguments

```
{0}proc(1) ::
command('Current . 'simulate('writer ; 'reader ;)) ;
(...) |
{0}proc(3) :: nil |
{0}proc(10) :: nil |
{0}proc(12) ::
unlock ;
```

A lot has happened. We have two more processors, processor 10 for the writer and processor 12 for the reader. The situation in the state is the following:

- Reference 11 points to object 11, which is the writer and belongs to processor 10.
- Reference 13 points to object 13, which is the reader and belongs to processor 12.
- Processors 3 and 10 are unlocked.
- Processor 1 no longer has the lock on processor 12 it obtained during the creation of the reader, but processor 12 is still locked until it executes its unlock operation.
- The environment for processor 1 now contains the variables writer (reference 11) and reader (reference 13).
- The channel has an `INTEGER` attribute with a value of -1 indicating that the channel is empty (the channel only takes natural numbers).

Interesting about this command is that it will need two locks for its arguments and one of those locks is not available yet.

```
eval(channel(174, 1, 0), 'Current) ;
eval(channel(174, 2, 2), 'writer) ;
eval(channel(174, 2, 1), 'reader) ;
wait(channel(174, 1, 0)) ;
wait(channel(174, 2, 2)) ;
wait(channel(174, 2, 1)) ;
call(channel(174, 1, 0) .data, 'simulate,
  (channel(174, 2, 2) .data ; channel(174, 2, 1) .data ;),
  ('writer ; 'reader ;)) ;
```

The first thing that has to happen when executing a command is of course the evaluation of the target and the arguments. There are no queries involved, so this is not interesting enough to be looked at in detail.

After the evaluation, a call is issued. We can see that some of the arguments to that call are still channels. After the evaluations are completed, the channels are replaced, as we see in step 184.

```
{0}proc(1) ::
call(ref(2), 'simulate, (ref(11) ; ref(13) ;), ('writer ; 'reader ;)) ;
```

As we can see, the `channel(...) .data` placeholders have been replaced by actual references. Let's look at the arguments of the call. The first argument of the call is obviously the target and the second the name the feature to call. The third is a list of the evaluated references of the arguments to the feature call. The original expressions are also added.

We already know how a call is expanded. But this time, the locking aspect is interesting. So let's see what the `checkPreAndLockRqs` operation does.

```
lock(({proc(10)} U {proc(12)})) ;
command('writer . 'start(nil)) ;
command('reader . 'start(nil)) ;
checkPostAndUnlockRqs(({proc(10)} U {proc(12)}),
  procedure 'simulate(...)) ;
return(channel(184, 1, 0), proc(1)) ;
```

For the rest of the example, all the `nop` operations are removed and the `provided` operations are evaluated as soon as possible to save paper. We can see that processor 1 needs to lock the two processors 10 and 12 before executing the start procedures on both writer and reader. The application operation determined this because of the two arguments. Processor 1 does not have the locks so it need to obtain them. Unfortunately, processor 12 is still locked so Maude has to execute the `unlock` operation on it first.

Two steps later, the two processors are locked and the commands can be executed. We skip a lot of steps to the point where the writer likes two write something but can not, because the reader has not emptied the channel yet.

3.10 Waiting on a separate call

After some time, we reached step 390. The channel currently contains the first value the writer has provided.

```
{0}proc(1) ::
issue(proc(12),
  apply(channel(207, 1, 0), ref(13), procedure 'start(...), nil,
    proc(1), [empty,empty]) ;
) ;
checkPostAndUnlockRqs(({proc(10)} U {proc(12)}),
  procedure 'simulate(...)) ;
return(channel(184, 1, 0), proc(1)) ;
checkPostAndUnlockRqs(empty, procedure 'make(...)) ;
return(channel(1, 1, 0), proc(0)) ;
unlock ; |
{0}proc(3) :: nil |
{0}proc(10) ::
command('Current . 'write('channel ; (42) ;)) ;
checkPostAndUnlockRqs(empty, procedure 'start(...)) ;
return(channel(199, 1, 0), proc(1)) ; |
{0}proc(12) :: nil
```

The reader did not even get the call and the writer is about to write the second value into the channel. ”Why did not the issue statement of processor 1 execute?” you may ask. That is because it also involves processor 12 and processor 12 is at the bottom of the list.

3.10.1 Evaluation of the precondition

At step 403, processor 10 is evaluating the precondition for its `write` procedure.

```
(...)
{0}proc(10) ::
lock({proc(3)}) ;
eval(channel(402, 1, 0), 'a_channel . 'can_write(nil)) ;
wait(channel(402, 1, 0)) ;
provided channel(402, 1, 0) .data then
  nop ;
else
  issue(proc(3),
    unlock ;
  ) ;
  popObtainedRqLocks ;
  yield ;
  checkPreAndLockRqs(proc(10), {proc(3)}, procedure 'write(...)) ;
end ;
command('a_channel . 'write('a_data ;)) ;
checkPostAndUnlockRqs({proc(3)}, procedure 'write(...)) ;
return(channel(399, 1, 0), proc(10)) ;
(...)
```

It will call the `can_write` function on the channel to make sure it can write into it.

3.10.2 Evaluation of a query

When evaluating the query `a_channel.can_write`, the operational semantics, in step produces this:

```
(...)
{0}proc(10) ::
eval(channel(405, 2, 0), 'a_channel) ;
wait(channel(405, 2, 0)) ;
call(channel(405, 1, 0), channel(405, 2, 0) .data, 'can_write, nil, nil) ;
result(channel(402, 1, 0), channel(405, 1, 0) .data) ;
wait(channel(402, 1, 0)) ;
(...)
```

It first evaluates the target by reading it from the environment. It then calls the function `can_write`, passing the target and no arguments. The `call` operator for functions differs in the additional channel argument from the call operator for procedures. This channel is then used in a `result` operations to bind the result of the function call to the result of the expression.

3.10.3 Calling a function

In step 411, processor 10 issued the feature application for the function `can_write` to processor 3:

```
(...)
{0}proc(3) ::
apply(channel(405, 1, 0), ref(4), function 'can_write(...),
  nil, proc(10), [{proc(3)},{proc(10)}]) ; |
{0}proc(10) ::
wait(channel(405, 1, 0)) ;
(...)
```

As we can see, processor 10 passed its locks, even though it does not need to. This is a change from the original semantics for some cases of separate callbacks. Since a query always involves a wait-by-necessity, this simplifies the query call inference rule.

The `apply` operator is the same for all features.

3.10.4 Returning a query

In step 438, the `can_write` function is almost done. It only needs to read the `Result` variable from the environment and execute a function return operator:

```
(...)
{0}proc(3) ::
read('Result, channel(411, 2, 0)) ;
return(channel(405, 1, 0), channel(411, 2, 0) .data, proc(10)) ; |
{0}proc(10) ::
wait(channel(405, 1, 0)) ;
result(channel(402, 1, 0), channel(405, 1, 0) .data) ;(...)
(...)
```

Two steps later, processor 3's action queue is empty again and processor 10 got the result:

```
(...)
{0}proc(3) :: nil |
{0}proc(10) ::
result(channel(402, 1, 0), ref(33)) ;
wait(channel(402, 1, 0)) ;
provided channel(402, 1, 0) .data then
  nop ;
else
  issue(proc(3),
    unlock ;
  ) ;
  popObtainedRqLocks ;
```

```
  yield ;
  checkPreAndLockRqs(proc(10), {proc(3)}, procedure 'write(...)) ;
end ;
(...)
```

Another step and the precondition evaluation is (almost) finished:

```
(...)
{0}proc(10) ::
provided ref(33) then
  nop ;
else
  issue(proc(3),
    unlock ;
  ) ;
  popObtainedRqLocks ;
  yield ;
  checkPreAndLockRqs(proc(10), {proc(3)}, procedure 'write(...)) ;
end ;
(...)
```

The provided clause now contains a reference, which according to the heap is pointing to a `BOOLEAN` that has the value `false`. It will then release its locks and yield.

3.10.5 Yielding

After moving forward to step 446, the yield operator is now on top of processor 10's action queue.

```
(...)
{0}proc(10) ::
yield ;
(...)
```

The yield rewrite law will now increment the priority number of the processor:

```
(...)
{1}proc(10) ::
(...)
```

This causes maude to move the processor at the end of the list. This does not affect non-deterministic choices when searching the state space, but in the case of step-by-step execution, this nudges Maude to prefer other processors. The reader can now read the value. It will then yield for the writer, which writes the second number into the channel. Because it does not have any other instructions, it stays dormant. The reader can then read the second value and is also finished.

3.11 Finishing

At the end, all the action queues are empty and the heap only contains the last added object and the object pointed to by the last imported reference, and everything these objects require. Maude is then unable to take another step and stands still at step 1362.

Chapter 4

Contributions to the semantics

This work resulted in several contributions to the original formal specification. By implementing it, problems surfaced that were easily missed before. Most of them are minor, but there were also some major problems which are listed below.

4.1 Callback detection

In the original semantics, a callback is detected by looking at the supplier locks (taken from [5]):

Issue Operation – Separate Callback

$$\frac{\begin{array}{l} q \neq p \\ \neg\sigma.\mathit{locks\_passed}(p) \\ \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \\ \sigma.\mathit{rq\_locks}(q).\mathit{has}(p) \vee \sigma.\mathit{cs\_locks}(q).\mathit{has}(p) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

This did not work in the case when the lock were passed over several separate calls. If, for example, the lock is passed from `p` to `q` to `r`, and then `r` calls back `p`, the system would stand still because neither the issue separate nor the issue separate callback rule can be applied.

The solution, was to replace the callback detection by looking at whether the locks of the supplier are passed and the client has the supplier's call stack lock. If this is the case, the supplier passed its locks along a feature call chain to the client, which is now calling back.

Issue Operation – Separate Callback

$$\frac{\begin{array}{l} q \neq p \wedge \sigma.\mathit{locks\_passed}(q) \wedge \neg\sigma.\mathit{locks\_passed}(p) \wedge \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \\ \neg\sigma.\mathit{locks\_passed}(p) \\ \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

Of course, the rules for separate issue operations, feature application, call operation and creation of object on a separate explicit processor had to be changed as well.

4.2 Lock passing check

The semantics checks that before applying a feature, the locks of the current processor are not passed. Unfortunately, this happened on the wrong state.

4.3 Once result channels in loops

The application rule for a fresh once function specifies that all assignments and create statements that set the `Result` variable are followed by two instructions which update the once result in the heap.

Application Operation – Non-Once Routine or Fresh Once Routine

$$\frac{\dots}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \dots, r_n), q, \overline{l}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \dots \\ \qquad \texttt{provided } f \in \textbf{FUNCTION} \wedge f.\mathit{is\_once} \texttt{ then} \\ \qquad\quad f.\mathit{body} \\ \qquad\qquad [\mathit{result} := y; \texttt{set\_not\_fresh\_with\_result}(f)/\mathit{result} := y] \\ \qquad\qquad [\texttt{create } \mathit{result}.y; \texttt{set\_not\_fresh\_with\_result}(f)/\texttt{create } \mathit{result}.y] \\ \qquad \texttt{else} \\ \qquad\quad f.\mathit{body} \\ \qquad \texttt{end}; \\ \qquad \dots \\ \qquad s_p, \sigma''\rangle\end{array}}$$

The rule uses a fresh channel and it looks correct. If the statements involving the result are executed in a loop, then at the first iteration, the channel placeholder is replaced by the value of the `Result` variable. On the second iteration, however, no replacement happens and the heap is updated with the same result value as before.

The solution was to introduce a new operation that updates the once result. This way, a fresh channel is created for every iteration of a loop:

Application Operation – Non-Once Routine or Fresh Once Routine

$$\frac{\ldots}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \ldots \\ \qquad \texttt{provided } f \in \textbf{FUNCTION} \wedge f.\mathit{is\_once} \texttt{ then} \\ \qquad\quad f.\mathit{body} \\ \qquad\qquad [\mathit{result} := y; \texttt{set\_not\_fresh\_with\_result}(f)/\mathit{result} := y] \\ \qquad\qquad [\texttt{create } \mathit{result}.y; \texttt{set\_not\_fresh\_with\_result}(f)/\texttt{create } \mathit{result}.y] \\ \qquad \texttt{else} \\ \qquad\quad f.\mathit{body} \\ \qquad \texttt{end}; \\ \qquad \ldots \\ \qquad s_p, \sigma''\rangle\end{array}}$$

Set Once Routine Not Fresh Operation – Function With Result

$$\frac{\begin{array}{l} f \in \textbf{FUNCTION} \wedge f.\mathit{is\_once} \\ \sigma.\mathit{envs}(p).\mathit{top}.\mathit{names}.\mathit{has}(\texttt{Result}) \\ a \; is \; fresh\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{set\_not\_fresh\_with\_result}(f); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{read}(\texttt{Result}, a); \texttt{set\_not\_fresh}(f, a.\mathit{data}); s_p, \sigma\rangle\end{array}}$$

Chapter 5

User's Guide

5.1 Prerequisites

5.1.1 Maude

The interpreter relies on Maude 2.6. Any later version may also work, but has not been tested because 2.6 is the latest version as of March 2012.

Maude can be downloaded on its homepage: http://maude.cs.uiuc.edu/

5.1.2 SCOOP executable structural operational semantics

The software can be downloaded from the SCOOP repository at ETH, which is currently moving due to the shutdown of origo.

The details of how the SCOOP syntax is mapped to Maude constructs are described in section 2.2.

5.2 Running an example

To run an example (in this case channel.maude), open a terminal and navigate to the directory where the SCOOP interpreter is. Then execute the following:

```
>maude src/SCOOP.maude examples/channel.maude
```

`maude` is the maude executable. The example to run has to be the second argument to maude. You should then be greeted with:

```
         \||||||||||||||||||/
       --- Welcome to Maude ---
         /||||||||||||||||||\
      Maude 2.6 built: Dec  9 2010 18:28:39
```

```
      Copyright 1997-2010 SRI International
       Thu Mar  8 15:58:32 2012
Advisory: redefining module SET.
Advisory: redefining module LIST.
Advisory: redefining module MAP.
Advisory: "COLLECTIONS.maude", line 54 (fmod SET-COMMON):
  collapse at top of s:Set U s:Set may cause it to match
more than you expect.
==========================================
rewrite [1] in SYSTEM :
(import default __create__invariant_end(...) ;
 __create__invariant_end(...) ; __create__invariant_end(...) ;
 __create__invariant_end(...) ;)
{ 'CHANNEL_SIMULATOR } . 'make
 .
rewrites: 259 in 0ms cpu (0ms real) (~ rewrites/second)
result Configuration: (__create__invariant_end(...) ;
 __create__invariant_end(...) ; __create__invariant_end(...) ;
 __create__invariant_end(...) ; __create__invariant_end(...) ;
 __create__invariant_end(...) ;)

|-
  {0}proc(0) ::
  call(ref(2), 'make, nil, nil) ;
  issue(proc(1),
    unlock ;
  ) ;
  popObtainedRqLocks ; |
  {0}proc(1) :: nil
,1,


state(
regions((
  proc(0) --> empty # proc(1) -->
{obj(2, __create__invariant_end(...), empty)}), proc(1), (
  proc(0) --> false # proc(1) --> true), (
  proc(0) --> true # proc(1) --> true), (
  proc(0) --> ({proc(1)} ;) # proc(1) --> nil), (
  proc(0) --> proc(0) # proc(1) --> proc(1)), (
  proc(0) --> nil # proc(1) --> nil), (
  proc(0) --> nil # proc(1) --> nil), (
  proc(0) --> false # proc(1) --> false)),
heap(ref(2) <-> obj(2, __create__invariant_end(...), empty), empty,
obj(2, __create__invariant_end(...), empty)),
store(empty), void, 3)
```

The terms looking like `__create__invariant_end(...)` are concealed classes. Features are also concealed to improve readability.

5.2.1 Action queues

The natural number in braces preceding every processor is the processor's priority. The action queues are sorted by ascending priority and then by ascending processor identifier. The execution prefers action queues that are higher up, in other words action queues with a lower priority number.

5.2.2 State

Below the action queues is the current state.

Regions The arguments of the regions are the following:

1. A map from each processor to all the objects it handles
2. The processor added last
3. A map indicating which request queues are locked
4. The same for the call stack (always true)
5. A map from each processor to the stack of lock sets with the obtained request queue locks
6. The same for the obtained call stack lock
7. The same for the retrieved request queue locks
8. The same for the retrieved call stack locks
9. A map indicating the passed lock state of each processor

Heap The arguments of the heap are:

1. A map from references to objects maintaining the connection between references and objects
2. A map from pairs of processor and name to references which saves the once results
3. The object added last

Store The store contains a map from processors to environment lists (stacks). The environments contain a map from variable names to references and the feature associated with the environment.

5.2.3 Execution

The first step is the initialization, which transforms the input program into the initial configuration. This is done immediately after Maude starts and without user interaction.

To take one or more steps, use

```
Maude>cont n .
```

with `n` being the number of steps to take. If `n` is omitted, the program runs to completion if possible.

Chapter 6

Developer's Guide

6.1 Project directories

src/ All the source code of the operational semantics

examples/ Examples written in Maude

doc/ Documentation (e. g. this document)

6.2 Source files

In `SCOOP.maude`, all source files necessary to run SCOOP are loaded in the order they are needed. It is not recommended to load files from any other place to avoid loading some of them multiple times.

Big modules all have their own source files, but smaller modules are sometimes put together if it makes sense: All the features (`PROCEDURE`, `FUNCTION`, `ATTRIBUTE`, `ROUTINE` and `FEATURE`) are in one file named `FEATURE.maude`.

The file named `CONFIGURATION.maude` contains the definition of the configuration and all the rewrite rules. It is reasonable to later split up the rewrite rules in more modules and files.

6.3 Code conventions

6.3.1 Module structure

Sorts and subsort declarations should be placed in an extra module called *ORIGINALMODULENAME*`-SORTS` if possible (as explained in section 2.3.2). Then for every sort that can reasonably be placed in a set, a view from `TRIV` to *MODULENAME*`-SORTS` with the same name as the sort should be placed before the module containing the operators, rewrite laws and so on.

This way, a code module can also use a set of its sorts, because the view only points to the sorts module and not to the code module, which would result in a circular relationship.

The code module should be split in up to seven parts:

1. Inclusions
2. Constructors
3. Operators
4. Variables
5. Memberships
6. Equations
7. Rewrite rules

This way, every equation, rewrite rule etc. can use any operation that is included or defined in the module.

6.3.2 Naming conventions

Modules and views

The operators should be organized in modules according to the sorts they operate on. The name of a module should be in `UPPERCASE` and using an underline symbol (_) to separate words. Trivial views (`TRIV`) should always be named after the sort they refer to. Other views should be named in a comprehensible way.

Sorts

Sorts should always be named in `CamelCase`. Not only is this the usual way of defining them in Maude, it also distinguishes them from the modules.

Constructors

There are two general ways to define a constructor.

Smaller constructors should simply use the prefix notation and add all the data fields as arguments. For example:

```
op proc : Nat -> Proc [ctor] .
op noProc : -> Proc [ctor] .
```

More complicated constructors are better off by using a specific mixfix notation that resembles the structure:

```
op
  environment
    variables _
    current _
  end : Map{Name, Value} Feature
    -> Env [ctor format(ni n++i d ni d n--i d)] .
```

This makes the analysis much easier because every argument has a description.

Sometimes a custom mixfix notation that does not fit in these two styles is best.

Operators

Operators should be named in lower `camelCase` to not rely on the underline symbol, which has a special meaning in operator definitions for the mixfix notation. If an operator is part of a sort, the operator should be defined in mixfix notation, as in the following operator:

```
op _.doSomething(_, _) : CurrentSort ArgumentSort ArgumentSort
  -> CurrentSort [prec 10] .
```

The precedence value is important to support chains of operators. Be aware that Maude needs a space or parentheses to separate between tokens. The following call would not work:

```
cs.doSomething(a, b)
```

Whereas these would work:

```
cs .doSomething(a, b)
(cs).doSomething(a, b)
```

Object creators are the operators that are used to create an object (the constructors in object oriented languages). These should be named in the following style:

```
op new OBJECT.make(_) : ArgumentSort -> Object .
```

Variables

Variable names should be short. It is recommended to simply use the initials of the sort name and a number or apostrophes to distinguish. Long variable names do not make the equations and rewrite rules easier to understand. An `AppleSet` (or `Set{Apple}`) would be named `as`; further variables of that type have the names `as'`, `as0`, `as1`, etc.

Chapter 7

Related work

7.1 SCOOP

This work is part of the ongoing SCOOP project at ETH Zürich. Eiffel Software is currently implementing SCOOP as an experimental feature in their Eiffel implementation called EiffelStudio.

The first publication about SCOOP [4] goes back to 1993. The book *Object-Oriented Software Construction* [1] by Bertrand Meyer contains a chapter about SCOOP. Piotr Nienaltowski's PhD thesis [6] describes the whole SCOOP system and is still the main reference for SCOOP.

This work is based upon the structural operational semantics of SCOOP [5] defined by Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer.

7.2 Executable structural operational semantics

Verdajo et al. [7] give examples of several operational semantics implemented in Maude. They solved intricate problems using the meta-level.

Chapter 8

Conclusion

8.1 Future work

8.1.1 Missing functionality

Typing environment

Because of time constraints, a complete typing environment was not implemented, only a subset. Most of the semantics could be implemented, with a slightly different conditions and the assumption that the supplied code is correct.

Explicit processor tags

Explicit processor tags are the only part where a typing environment is important. Once it is in place, the processor tags can be easily added.

Asynchronous postcondition evaluation

Because of the work on exceptions in SCOOP, Benjamin Morandi and Bertrand Meyer determined that asynchronous postcondition evaluation is useless and implementing it in the interpreter would be pointless.

8.1.2 Improving the interpreter

The executable semantics is not complete. There are a few minor features missing.

Arrays

At the moment, the base library supports INTEGER and BOOLEAN. With the support of arrays, more examples are possible.

Scheduling

Using Maude meta-strategies are possible. Currently, the interpreter uses the fair rewrite facilities of Maude.

8.1.3 Front-End

A parser that parses the original syntax and generates the intermediate representation would make writing new examples much easier.

8.1.4 User interface

Working directly in Maude is cumbersome. It would be very nice to have a graphical user interface that would sit on top of Maude and provide an easy way to go step by step through the program.

State explorer

The user interface can parse after every step the current state and provide a graphical front-end to enable the user to easily find the parts of the state he needs. As an extra, the user could manipulate the state and then go on with the execution.

Moving backwards

By saving the state and the action queues at every point in the execution, the user interface can move backwards in time by simply replacing the current state with a saved one.

Pretty Printing

Instead of showing the ugly and bloated intermediate representation, the user interface could pretty print the output.

8.1.5 Improvements to the code quality

Some of the functions and constructors are not optimally implemented. Especially the regions constructor could be implemented leaner. The obtained call stack map can be removed because its content is static. All maps that map to a boolean value can be replaced by a set containing only the keys that map to true. Finally, the current prefix notation used for the constructors of the state, regions, store and environment is not optimal for readability.

Bibliography

[1] *Object-Oriented Software Construction*. Prentice Hall, 1997.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual. Available from: http://maude.cs.uiuc.edu/maude2-manual.

[3] Maude homepage. Available from: http://maude.cs.uiuc.edu.

[4] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 1993.

[5] B. Morandi, S. Nanz, and B. Meyer. A comprehensive operational semantics of the scoop programming model. *Arxiv preprint arXiv:1101.1038*, 2011.

[6] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zürich, 2007.

[7] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.