Babelsberg/RML: Executable Semantics and Language Testing with RML

Tim Felgentreff, Robert Hirschfeld, Alan Borning, Todd Millstein

Technische Berichte Nr. 103

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 103

Tim Felgentreff | Robert Hirschfeld | Alan Borning | Todd Millstein

Babelsberg/RML

Executable Semantics and Language Testing with RML

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2016
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URN urn:nbn:de:kobv:517-opus4-83826
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-83826

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-348-0

New programming language designs are often evaluated on concrete implementations. However, in order to draw conclusions about the language design from the evaluation of concrete programming languages, these implementations need to be verified against the formalism of the design. To that end, we also have to ensure that the design actually meets its stated goals. A useful tool for the latter has been to create an executable semantics from a formalism that can execute a test suite of examples. However, this mechanism so far did not allow to verify an implementation against the design.

Babelsberg is a new design for a family of object-constraint languages. Recently, we have developed a formal semantics to clarify some issues in the design of those languages. Supplementing this work, we report here on how this formalism is turned into an executable operational semantics using the RML system. Furthermore, we show how we extended the executable semantics to create a framework that can generate test suites for the concrete Babelsberg implementations that provide traceability from the design to the language. Finally, we discuss how these test suites helped us find and correct mistakes in the Babelsberg implementation for JavaScript.

1. Introduction

Babelsberg is a family of object-constraint languages. Recently, we have developed a formal semantics to clarify some issues in those languages, in particular pertaining to object identity, types, object structure, and allowed methods in constraints [6]. Our aim in developing a formal semantics for Babelsberg has been primarily a practical one, including providing a guide for language implementors and proving useful language properties that then can be relied on by programmers. One piece of evidence of success is that we have in fact clarified the desired behavior of the language and have adapted the existing implementation in JavaScript, Babelsberg/JS, to conform to it. As we evolved our semantics, we developed a set of 47 test programs that demonstrate different issues we address.

This report is written as supplementary material to the paper on the formal semantics, and should be considered an in-depth look at the implementation of the executable semantics, rather than a document that stands on its own. We report on our findings in creating an executable semantics in RML [10]. Concretely that means:

- We provide a mapping from the solver language in our semantics into a concrete solver language. While in our semantics we treat the solver as a black box, this is not possible for a concrete executable. We used a branch of the Z3 solver [3] that supports soft constraints to run our example programs.
- We describe a framework to translate our test programs into the language of concrete implementations, which we have used to generate test suites for the implementations in JavaScript, Ruby, and Smalltalk. This gives language implementers an easy mechanism to check their conformance to the semantic test suite.
- We discuss in-depth some existing programs in those languages that needed to be changed to work with the new semantics. We evaluate the changes and how we think they contribute to make the program easier to understand. We also discuss programs that do not work anymore because they use constructs we explicitly decided to disallow.

2. An Executable Semantics with a Black-Box Solver

RML is a programming language to generate executables from big-step semantics. It provides *relations* as basic building blocks for the semantics and translates these to C. It was possible to implement our semantics exactly in RML. As the semantics does not model the solving process, we have added a translation from our constraint syntax to Z3, which solves the constraints and generates a model of the new heap and environment. Because our semantics relies on soft constraints, we use an experimental version of Z3 that supports them [1]. Our code is publicly available on GitHub at `https://github.com/babelsberg/babelsberg-rml` (as of Apr 25, 2016).

To develop the executable semantics for Babelsberg/Objects three steps are required.

a) We must create an abstract syntax definition for RML and a parser to generate the abstract syntax tree in that definition.

b) We must implement all the judgments and their rules in the formal semantics as RML *relations* and corresponding *rules*.

c) The judgment in the formal semantics that represents a call to the solver does not have a rule – the formal semantics treat the solver as a black box. We must provide a concrete solver and implementation to call it to make an executable semantics.

2.1. RML Syntax

The abstract syntax of our language can be mapped almost directly. Most of the work is done using a hand-written Yacc grammar whose productions call the appropriate RML token generators. These are generated from the *abstract syntax definition* given in the RML semantics. The syntax we have given in our semantics for Babelsberg/Objects is:

Statement	s	::=	skip | L := e | always C | once C
			| s;s | if e then s else s | while e do s
Constraint	C	::=	ρ e | C $\wedge$ C
Expression	e	::=	v | L | e $\oplus$ e | I | o | new o | D
			| e.l(e_1,...,e_n)
Identity	I	::=	L == L
Object Literal	o	::=	{l_1:e_1,...,l_n:e_n}
L-Value	L	::=	x | L.l
Constant	c	::=	true | false | nil | base type constants
Variable	x	::=	variable names
Label	l	::=	record label names
Reference	r	::=	references to heap records
Dereference	D	::=	H(e)
Method Body	b	::=	s; return e | return e
Value	v	::=	c | r | {l_1:v_1,...,l_n:v_n}

This translates to the following abstract RML syntax

```
module babelsberg:
(* Abstract syntax for the BabelsbergP language *)
  datatype Program = PROGRAM of Statement

  datatype Statement = SKIP
                     | ASSIGN of LValue * Exp
                     | ALWAYS of Constraint
```

```
                | ONCE of Constraint
                | SEQ of Statement * Statement
                | IF of Exp * Statement * Statement
                | WHILE of Exp * Statement
  datatype Constraint = CONSTRAINT of Rho * Exp
                 | COMPOUNDCONSTRAINT of Constraint * Constraint
  datatype Exp = VALUE of Value
            | LVALUE of LValue
            | OP of Exp * Op * Exp
            | IDENTITY of Exp * Exp
            | IRECORD of ObjectLiteral
            | UIDRECORD of ObjectLiteral
            | DEREF of Dereference
            | CALL of Exp * Label * Exp list
  (* Identity folded into Exp *)
  type ObjectLiteral = (Label * Exp) list
  datatype LValue = VARIABLE of Variable | FIELD of Exp * Label
              | ASSIGNDEREF of Dereference (* just for the parsing *)
  datatype Constant = TRUE | FALSE | NIL | REAL of real | STRING of string
  type Variable = string
  type Label = string
  type Reference = int
  type Dereference = Exp
  datatype MethodBody = METHOD of Statement * Exp | SIMPLE of Exp

  datatype Value = K of Constant | O of ObjectLiteral | R of Reference

(* Helper types *)
  datatype Rho = WEAK | MEDIUM | REQUIRED
  datatype Op = ADD | SUB | MUL | DIV
             | LESSTHAN | LEQUAL | EQUAL
             | NEQUAL | GEQUAL | GREATERTHAN
             | AND | OR

(* Type syntax *)
  datatype Type = PRIMITIVE | TRECORD of (Label * Type) list

(* Bindings and environments *)
  type Env = (Variable * Value) list
  type Scope = (Variable * Variable) list
  type Heap = (Reference * ObjectLiteral) list
  type Cstore = (Scope * Constraint) list
  type Istore = (Scope * Constraint) list

  relation evalprogram: Program => ()
  relation eval: (Env, Scope, Heap, Cstore, Istore, Exp) => \
        (Env, Heap, Cstore, Istore, Value)
  relation alert: (string list, Exp list, string list) => ()
end
```

As you can see, RML adds a root production `Program` (line 3) and a number of productions to resolve to the set of operations and constraint strengths (lines 38–41). Notably, things like the Environment and Heap, that are not part of the formal syntax, but that we refer to in the rules are also part of the abstract RML syntax (lines 47–51). Finally, we declare (but do not define) the main rules for evaluation (lines 53–55) and an `alert` relation to print debugging statements. Relations map an input tuple to an output tuple, and use rules to pattern match which operation to perform. We define these relations here so that RML generates their declarations into a C header, which we can then use to parse the program with Yacc, evaluate single expressions, and print debugging information from C.

2.2. RML Semantics

Similarily, the semantics rules map directly to rules in RML. RML evaluates such rules through unification of variables and backtracking until finding an appropriate rule. If all options are exhausted and we would get "stuck" in the formal semantics, RML halts the execution. RML syntax is modeled to correspond closely to the formatting of operational semantics rules. We will give just one example:

$$\frac{\begin{array}{c}\langle E|S|H|C|I|e\rangle \Downarrow \langle E'|H'|C'|I'|r\rangle \\ H(r) = \{l_1{:}v_1,\ldots,l_n{:}v_n\} \qquad 1 \leqslant i \leqslant n\end{array}}{\langle E|S|H|C|I|e.l_i\rangle \Downarrow \langle E'|H'|C'|I'|r\rangle} \qquad \text{(E-Field)}$$

In RML syntax, this becomes:

```
1 rule eval(E,S,H,C,I,e) => (E',H',C',I',R(r)) &
2       alert(["E-Field(", l, ")\n"], [], []) &
3       Util.lookupHeap(H, r) => fvalues &
4       Util.lookupRecord(fvalues, l) => v
5       ----------------------------------------------------
6       eval(E,S,H,C,I, LVALUE(FIELD(e, l))) => (E',H',C',I',v)
```

Line 1 represents the recursive call to evaluate the left-hand side of the field access. The result is then matched against the intended result. This line succeeds only if RML can unify the evalution of `e` and `R(r)`. Since we expect the result to be a reference, we pattern match it against a box of type `R`, and unify its contents with the variable `r`. RML matches rules in order.

Line 2 "matches" a debug print. When the first rule succeeds, we proceed to this rules, which always succeeds.

Line 3 proceeds to lookup `r` on the heap. We use a utility function for this that simply searches the set of reference - object pairs on the heap. When found, the object is bound to the variable `fvalues`.

Line 4 looks up the intented label on the bounds to `fvalues` by the previous rule. This looks slightly different from the semantic rule. In the rule, we enumerate all labels and check that the label we want is within the range of enumerated labels. In practice we allow any string as a label, and do not generate a total order, so the field lookup iterates over the fields in the object and tries to match the labels using string equality.

Line 6 finishes the evaluation. If all rules could be matched, the rule succeeds, and the evaluation finishes and returns the updated environment, heap, constraint and identity-constraint stores, and the value of the field.

2.3. The Solver Judgment

Given the above implementation of the syntax, and the implementation of the remaining semantic rules, RML is able to generate a binary that parses and begins to execute example programs in Babelsberg/Objects. So suppose we have a program `x := 1 + 2`, RML will evaluate the right-hand side of this expression and then continue to execute the assignment rule. Assignment will execute to create a set of constraints and then hand those to the solver – for which the formal semantics provides no rules, only this judgment:

$\boxed{\mathtt{E;H} \models \mathtt{C}}$

Our implementation has to implement this judgment which involves:

1. Converting the constraints from the syntax of the formal semantics into a form that can be understood by a solver
2. Running the solver and retrieving its result
3. Parsing the result and generating a new environment and heap to return to the executable, and continue evaluating the rule through RML

2.3.1. Converting Constraint Syntax

For our executable semantics, have chosen Z3 as a solver, because it supports the types of constraints we want: constraints over integers, booleans, reals, records, and finite domains of uninterpreted symbols. Z3 can use SMT2 as input specification, an open format for constraint system definition. In SMT2, we have to declare each type that we will use (including exhaustively listing the values for finite types), declare each variable and its type that will appear in a constraint, and then declare the constraints.

Declaring Types and Variables Z3 type checks all expressions and does not allow variables to change their type. Our structural typing rules, however, do allow variables to change their type, for example, from a real to an integer or a string. In the implementation we thus have to reconcile Z3 typing with our structural typing rules. To do so, we declare variables in Z3 as a union type `Value` that has real, boolean, reference, and value class components[1]. The latter is encoded as a Z3 array that maps *labels* to reals. Labels and reference are declared as finite domain types that range over the existing references and record labels respectively. Note that we only support value classes with real fields here. The reason is that Z3 does not support recursive structures through arrays. This is not a problem in practice, as we can explicitly define datatypes for more nested records up to a finite depth K as required – for simplicity, we give here only a definition for $K = 1$:

```
(declare-datatypes () ((Value (Record (rec (Array Label Real)))
                        (Real (real Real))
                        (Bool (bool Bool))
                        (String (string Real))
                        (Reference (ref Reference)))))
```

Objects, analogous to value classes, are represented as Z3 arrays that map labels to values. Because arrays in Z3 are total functions, and the labels form one global finite set, we declare a special `invalid` reference as the default value for all object fields. Fields that only have the invalid reference are then ignored when reading the model from the solver:

```
; A default record has 'invalid' for all fields
(declare-const iRec (Array Label (Value)))
(assert (= iRec ((as const (Array Label (Value))) (Reference invalid))))
```

We must furthermore declare the global heap and environment. The global environment is naturally the mapping of the variables to values that Z3 produces. The heap we declare as a mapping from references to records. Again, because mappings in Z3 have to be complete, we must assert that any invalid value is mapped to an invalid record:

```
; Records are (Array Label (Value))
(declare-fun H ((Value)) (Array Label (Value)))
(assert (and (= (H (Reference invalid)) iRec) (= (H (Reference nil)) iRec)))
```

Finally, we must declare the finite domain types `Reference` and `Label`. To determine their domains, we again provide rules that match all possible expressions, and construct a list of the necessary definitions for those rules that matter. The relations to construct the list of strings are defined like this:

```
relation pRefDomE: babelsberg.Exp => string list
relation pLabDomE: babelsberg.Exp => string list
```

Most rules for these two relations simply recurse and add nothing to the list. The only rules that adds a string for references is the one that processes a direct refer-

[1] Unfortunately, at the time of this writing the optimizing strategy for Z3 produces different results, depending on the ordering of the alternatives in the union type. For a variable that only has a soft constraint that would make it an array, Z3 will sometimes (depending on the order of the union) make it a real. We work around this by trying with another ordering, which then produces the correct result.

ence (remember that constraints as passed to the solver will only have references as part of objects or as part of equations in identity constraints):

```
rule int_string(r) => rs &
     string_append(" ref", rs) => rs'
     ----------------------------------------------------
     pRefDomE(babelsberg.VALUE(babelsberg.R(r))) => [rs']
```

Here, the integer-type reference `X` is transformed into a unique string `refX` and added to the list. The important rules for labels are the ones that include field access, value objects, and heap objects:

```
rule printLabContents(ary) => str
     ----------------------------------------------------
     pLabDomE(babelsberg.VALUE(babelsberg.O(ary))) => str

rule printLabContents(ary) => str
     ----------------------------------------
     pLabDomE(babelsberg.IRECORD(ary)) => str

rule pLabDomE(e) => es &
     string_append(" ", l) => ls &
     list_union(es, [ls]) => es'
     ----------------------------------------------------------
     pLabDomE(babelsberg.LVALUE(babelsberg.FIELD(e, l))) => es'
(* ... *)
relation printLabContents: babelsberg.ObjectLiteral => string list =
  axiom printLabContents([]) => []

  rule pLabDomE(e) => es &
     printLabContents(rest) => rests &
     string_append(" ", l) => ls &
     list_union(es, [ls]) => fl &
     list_union(fl, rests) => str
     ---------------------------------------
     printLabContents((l, e) :: rest) => str
end
```

The rules use a helper relation `printLabContents` to extract all labels from a record and add them to a list. The final declarations for labels and references look like this:

```
(declare-datatypes () ((Label undef label1 label2)))
(declare-datatypes () ((Reference invalid nil ref1 ref2)))
```

Once we have all these, we can declare all variables that appear in constraints. Again, we use a relation that recurses and constructs a set of variable declarations from the given constraints. The only rule that is (besides recursion) non-empty is:

```
rule printFieldVar(x) => xs &
     string_append_list(["(declare-const ", xs, " (Value))\n"]) => str
     -----------------------------------------------------------------
     pDefsE(babelsberg.LVALUE(babelsberg.VARIABLE(x))) => [str]
```

Declaring Constraints Converting our constraints into SMT2 is straightforward – for each form possible in constraints, we create a rule that converts the form into a string. For this, we use the relations `printZ3C` and `printZ3E`. For example, to print a required constraint:

```
1  rule printZ3E(e) => es &
2       string_append_list(["\n(assert (bool ", es, "))"]) => cs
3       ------------------------------------------------------------
4       printZ3C(babelsberg.CONSTRAINT(babelsberg.REQUIRED, e)) => cs
5  (* ... *)
6  rule printZ3E(e1) => e1s &
7       printZ3E(e2) => e2s &
8       string_append_list(["(equal ", e1s, " ", e2s, ")"]) => es'
9       ---------------------------------------------------------
10      printZ3E(babelsberg.IDENTITY(e1, e2)) => es'
```

The rule on lines 1–4 matches a required constraint, and prints it as a Z3 expression using, for example, the rule in lines 6–10. Similar to the prior rules that create variable and domain declarations, these rules simply traverse the compound constraints generated by the inlining judgments, each rule generating a part of the final SMT2 constraint expression for Z3.

For soft constraints, we use the optimization solver in a development version of Z3, which adds the ability to assert preferential, non-required assertions. Weak constraints thus are translated to such non-required assertions. For example, the following stay constraint is translated as shown:

```
# weak x0 = 4.0
(declare-const x0 (Value))
(assert-soft (bool (equal x0 (Real 4.0))) :weight 1)
```

Finally, to encode strings in Z3, we use an encoding into reals. This encoding is really just a prefix (1), and then decimal coded character values as they appear in the string, padded to 3 digits. Z3 can only perform equality operations on strings encoded in this manner, which is enough for our purposes:

```
rule string_list(c) => clist &
     list_map(clist, char_int) => ilist &
     list_map(ilist, int_string) => islist &
     list_map(islist, string_padding3) => padded_list &
     string_append_list(padded_list) => isstring &
     string_append_list(["(String 1", isstring, ")"]) => s
     ----------------------------------------------------
     printZ3E(babelsberg.VALUE(babelsberg.K(babelsberg.STRING(c)))) => s
```

2.3.2. Running the Solver

RML generates C code from the semantic definitions. This makes it easy to drive it from Yacc, but also allows us to add “primitive” operations to RML itself. In our case, this primitive operation is the call to the solver. After the transformation is complete, we pass the entire SMT2 string that was generated to the `solve` relation, which is implemented in C. This relation simply writes the SMT2 string to a file and prints it to the terminal. It then waits until it receives a solution in the assignments syntax. This is useful both to review the generated constraints, as well as check the solution that Z3 comes up with against what we expected. The rule does not itself call Z3, so the solution can be supplied by the user for testing. The solution is then parsed again using the Yacc grammar as a set of assignments which are converted in RML into a new environment and heap. Thus, a solution is itself simply a set of assignments to variables which is run (without again calling the solver).

2.3.3. Parsing the Solver Output

In the technical report of our formalization, we provide 47 test programs that illustrate various concerns of the semantics. Of those 47 programs, three use constraints on strings, which are not supported in the version of Z3 we used (a string theory exists for an old version of Z3, but does not work in recent versions [11]). The remaining 44 all run and produce the expected results at each step of the execution. To run those tests automatically, the third required step is a translation from the output of Z3 to the Babelsberg/Objects assignments syntax. For this, we use a hand-written Ruby script that transforms the Z3 output into the correct syntax. Since the assignments are always simple value assignments (assigning an object to a heap location, or a reference to a variable), regular expressions suffice for this task.

As an extension to the RML implementation, we add a module to transform the example code into test cases for the various implementations of Babelsberg. This may provide an avenue for future development of Babelsberg – as the semantics evolve, test cases for the existing implementations can be generated and the implementations adapted to pass the generated tests. To generate the tests, we run our examples through the generated RML executable, translate them into the syntax of the target language using a simple mapping provided for each language, and wrap them in a test scaffold (also specific to each language) to execute them in each implementation.

3. Generating a Language Test Suite from the Semantics

It is not trivial to check implementations for conformance to the semantics. This is regardless of whether the implementation evolved before or alongside the formal semantics or whether the implementation was created following the principles of the formalization. In either case, language specifics and optimization will add complexity and make the conformance to the semantics harder to judge.

Many languages, such as Ruby[2], Perl[3], or Java[4] provide an extensive test suite to check concrete implementations for conformance to a (sometimes informal)

[2] `http://rspec.info/`, retrieved Feb 25, 2015.
[3] `http://perl6.org/specification/`, retrieved Feb 25, 2015.
[4] `http://openjdk.java.net/groups/conformance/`, retrieved Feb 25, 2015.

specification. Babelsberg, however, is not a language design, but a design for a family of languages (of which Babelsberg/Objects may be seen as one). Thus, to provide test suites for concrete implementations, we need a way to transform the tests we have into the host language of the concrete implementation.

Our basic test suite is written in the language of Babelsberg/Objects. The tests evolved alongside the semantics, and we adapt and extend the suite when adding new features to the design. Given our executable semantics, we can immediately check if a change works as intended. To get the same kind of quick feedback in the actual implementations, our executable semantics repository includes a framework to generate language-specific test suites. A new language implementer only needs to provide two files: a scaffolding file that sets up the language specific testing environment and includes a special placeholder which can be replaced by the framework with the concrete test case code.

The important things every implementation's scaffold needs are:

- An implementation of the methods used in the semantics test cases available for calling from the tests, in particular for testing field equality.
- A mechanism to create records with variable numbers of fields.
- Any setup code the solvers used in the language require.
- The macro `INSERTHERE` at the beginning of the line on which to insert the generated test case code.

For example, the scaffolding of the test cases for Babelsberg/JS looks like this:

```
TestCase.subclass('babelsberg.testsuite.SemanticsTests', {
 fieldEquals: function(o1, o2) {
  if (!o1) return false;
  if (!o2) return false;
  for (var key in o1) {
   if (key[0] !== "$" && key[0] !== "_") {
    if (typeof(o1[key]) == "object") {
     this.fieldEquals(o2[key], o1[key]);
    } else {
     if (o1[key] !== o2[key]) return false;
    }
   }
  }
  for (var key in o2) {
   if (typeof(o2[key]) == "object") {
    this.fieldEquals(o2[key], o1[key]);
   } else {
    if (o1[key] !== o2[key]) return false;
   }
  }
  return true;
 },
 addPt: function(self, other) {
  return this.Point(null, self.x + other.x, self.y + other.y);
  },
  divPtScalar: function(self, scale) {
    return this.Point(null, self.x / scale, self.y / scale);
  },
  ptEq: function(self, other) {
    return self.x == other.x && self.y == other.y
  },
/* ... */
INSERTHERE

});
```

The second file a developer has to provide is an RML specification that can transform the source of a test program into the syntax of the target language. This is highly language dependent, but straightforward. Each evaluation rule in the basic semantics – for the entire program, for statements, expressions, objects, and the operations on them – needs a rule that translates this operation into the right syntax. How the developer names these relations is not important. The framework does expect the developer to include two things, however: the original Babelsberg/Objects semantics, as well as an assertions wrapper. These are used to run the program in the source transformation process and create the appropriate assertions each time the solver is called. The two entry point relations the framework expects are `printprogram` and `printassert`. For Babelsberg/JS, these declarations look as follows:

```
module javascript:
  with "../objects/babelsberg.rml"
  with "assertions.rml"

  relation printprogram: babelsberg.Program => ()
  relation printassert: babelsberg.Program => ()
end
```

The first relation is called to start executing and transforming the program. The second assertion is called with the output from the solver transformed into assignments – the language implementer must take care that these are transformed into the appropriate assertions. The implementation for JavaScript is:

```
relation printassert: babelsberg.Program => () =
  rule print " this.assert(" &
       printE(babelsberg.LVALUE(l)) & print " === " & printE(e) &
       print ");\n"
       -------------------------------------------------------
       printassert(babelsberg.PROGRAM(
                   babelsberg.ASSIGN(l,babelsberg.DEREF(e))))

  rule print " this.assert(this.fieldEquals(" &
       printE(babelsberg.LVALUE(l)) & print ", " &
       printE(babelsberg.IRECORD(fieldexps)) & print "));\n"
       ------------------------------------------------------------------
       printassert(babelsberg.PROGRAM(
                    babelsberg.ASSIGN(l,babelsberg.IRECORD(fieldexps))))

  rule print " this.assert(" &
       printE(babelsberg.LVALUE(l)) & print " == " & printE(e) &
       print ");\n"
       ----------------------------------------------------------
       printassert(babelsberg.PROGRAM(babelsberg.ASSIGN(l,e)))
end
```

As you can see, the assertion printing only has to deal with very few cases. Furthermore, note that we simply `print` the transformed code (rather than collecting it in a string). The framework takes care to redirect the standard output of the transformation executable and insert it into the scaffold.

Since the transformation consists of RML rules itself, it is itself an executable semantics, with the added side-effect of translating the executed program. Since the executable semantics for Babelsberg/Objects and the translation process for each language are thus tied together, the test suites for the concrete languages can be updated whenever the semantics change or a test is added. As an example, consider the following test from our formal semantics that is supposed to fail on the last line:

```
x := 0;
y := 0;
always medium x=10;
always y=test(x);
```

Translating just this test into JavaScript yields the following:

```
module('babelsberg.testsuite').
requires('lively.TestFramework',
      'babelsberg.constraintinterpreter',
      'z3.CommandLineZ3').
toRun(function() {
  TestCase.subclass('babelsberg.testsuite.SemanticsTests', {
    Test: function(self, i) {
      ctx = {i: i};
      always: {
        priority: "medium",
        ctx.i == 5;
      }
      return ctx.i + 1
    },
    test1: function() {
      bbb.defaultSolvers = [new CommandLineZ3(), new DBPlanner()];
      bbb.defaultSolvers[1].$$identity = true;
      var ctx = {unsat: false};

      try {
        ctx.x = 0.0;
      } catch (e) { ctx.unsat = true }
      this.assert(ctx.x == 0.0);
24    try {
25      ctx.y = 0.0;
26    } catch (e) { ctx.unsat = true }
27    this.assert(ctx.x == 0.0);
28    this.assert(ctx.y == 0.0);
29    try {
30      always: {
31        priority: 'medium';
32        ctx.x == 10.0
33      }
34    } catch (e) { ctx.unsat = true }
35    this.assert(ctx.x == 10.0);
36    this.assert(ctx.y == 0.0);
37    try {
38      always: {
39        priority: 'required';
40        ctx.y == this.Test(null, ctx.x)
41      }
42    } catch (e) { ctx.unsat = true }
43    this.assert(ctx.unsat == true);
44  }
45 });
46 });
```

Lines 1–15 are part of the scaffold. We provide the global function `test` from the semantics as a `Test` method on the TestCase object itself (ll.8–15). The `test1` method is automatically translated from the example, and first sets up the default solvers to be used (l.17), Z3 for value constraints and DeltaBlue for identity constraints (l.18). It then sets a local `unsat` flag to false, to indicate that no solving failed, yet (l.19). Each assignment and constraint definition is guarded against failure and sets this flag to true (ll.23, 27, 35, 43). After each call to the solver, assertions are generated from the default solution (ll.24, 28–29, 36–37). If we expect the solver to fail, we assert that an exception was caught and the handler set the `unsat` flag (l.44).

4. Applying the Semantics to Babelsberg/JS

All of the existing implementations of Babelsberg must be adapted to pass the tests, because they all deviate from our proposed semantics in one way or another. We have modified the JavaScript implementation, as it is the most advanced. After modification, of the 44 test cases that work in the RML semantics, 41 produce the expected results. The remaining three all specifically test properties of value classes in constraints. Since JavaScript does not support value classes (and we do not emulate them in some way in the scaffolding code) these three tests fail, in one case producing no result when there is a solution, and the other two producing a solution when solving should fail. We have also tested, but not adapted, the Ruby and Squeak implementations. Both Babelsberg/R and Babelsberg/S pass 28 out of 44 tests. In addition to the value class failures, these implementations also fail some tests concerning object identity, have insufficient support for explicit identity constraints, and support only an older version of Z3 that does not include soft constraints. A proof-of-concept Python implementation exists, but was not tested.

Adapting Babelsberg/JS to follow the semantics presented here required changes to both to the translation from imperative expressions to constraints (called *constraint construction* in Babelsberg/JS) and to the way in which the solvers are called. Additionally, we had to update some of the existing example programs that break with the new semantics.

4.1. Modifications to the Implementation

Adapting Babelsberg/JS to follow the semantics presented here required changes to the translation from imperative expressions to constraints to honor the restrictions on methods in constraints and to enforce the separation of value and identity constraints.

Restrictions on Methods In Constraints First, we modified the code that translates JavaScript expressions into constraints to not inline methods that have more than a simple return expression. Babelsberg/JS uses a source transformation to translate *BabelsbergScript* into JavaScript. The only syntactic addition is the syntax for declaring constraints:

```
always: { pt.x == pt.y }
```

The above code snippet is valid JavaScript: it declares a label `always` and a block of code. Our source transformation picks out all blocks attached to an `always` label, and translates it into this:

```
bbb.always(
  {
   solver: bbb.defaultSolver,
   ctx: {pt: pt}
  },
  function() {
     return pt.x == pt.y;
  }
);
```

The `bbb.always` takes the passed function and interprets it in a custom interpreter mode to create the constraint system. In doing so, it inlines any methods and operations in them. To follow the semantics, we simply check when inlining if the method body includes anything else besides a return statement. Methods that do are executed and their result returned, so they can only be used in the forward direction.

Babelsberg/JS does not prohibit creating new objects in constraints – we allow these since JavaScript does not include value classes, and assume that, when new objects are created in constraints, our conventions on object creation, modification, and not testing for identity are followed.

In the semantics, constraints a inlined whenever we need to solve. As an optimization, Babelsberg/JS omits re-translating expressions if no dependent variables or methods used therein have changed. We left this optimization in place, expecting that the test suite would show failures if this is not a legal interpretation of the semantics.

Separating Identity from Value Constraints The second change adds support for our restricted form of identity constraints. Before, users of Babelsberg/JS could use identity checks as part of ordinary constraints, as long as they used an appropriate solver. We now disallow the identity equality operation as part of constraints, and add the restricted form of identity constraints presented in our semantics. Thus, when an identity relation comes up as part of a constraint expression (even if within a called method), unless it is the *only* type of operation, we generate an exception. Thus, the following is still valid for a given a and b:

```
function identical(obj1, obj2) { return obj1 === obj2 }
always: { identical(a, b) }
```

But it would fail if we were to define identical like this:

```
function identical(obj1, obj2) {
  return obj1 == obj2 || obj1 === obj2
}
```

In the executable semantics, identity constraints can be solved using Z3 by reasoning over the finite domain of available references and updating the environment. However, in JavaScript we cannot reflect on the heap or use first-class references, so identity constraints cannot be solved in the same way. Instead, we use the DeltaBlue local propagation solver [7] to propagate identity changes as they occur, and call any solvers for value constraints after that.

In the semantics, identity and value constraints are solved separately. If we were to hand them to the solver all at once, we cannot control if identity changes occur to satisfy value constraints. Babelsberg/JS, through its implementation of a cooperating solvers architecture, already supports solving in multiple phases. The implementation follows the design by Borning et al. [2], which assigns constraints to different *regions*. These regions are ordered and solved one after another, with variables in earlier regions appearing as read-only in later regions. In Babelsberg/JS, regions correspond to solvers, and are simply ordered by an adjustable *priority* on the solver instance itself. The two-phase solving of identity and value constraints in the formalism is thus implemented simply by ensuring that the DeltaBlue solver for identity constraints is always run before any other solvers. We do this by using a framework-private DeltaBlue instance with the highest possible priority. Note in particular that this still allows the developer to use other instances of the DeltaBlue solver for value constraints, but, just as with all solvers for value constraints, they will be called after all the object identities are determined.

4.2. Modifications to Existing Programs

Our changes break a number of existing programs in Babelsberg/JS. These programs are written for the LivelyKernel [9] JavaScript environment and use many methods from its library.[5]

Argument Checking One issue we encountered is that many LivelyKernel methods have, besides a return statement, some statements that check the number, types, or structure of arguments. As per our semantics, such methods do not work multi-directionally. In fact, allowing them to be truly multi-directional would allow the solver to change the number, types, or structure of arguments, which is the kind of surprising behavior we want to avoid. As an example, consider the frequently used method to add two points in LivelyKernel:

```
function addPt(p) {
  if (arguments.length != 1) throw ('addPt() only takes 1 parameter.');
  return new lively.Point(this.x + p.x, this.y + p.y);
}
```

In Babelsberg/R, we do include an experimental solver that can solve for the length of arrays. While in this method it is not an issue, other methods dispatch based on the numbers of arguments passed. One such method is the method to make a rectangle:

```
makeRectangle: function (/**/) {
  var bounds;
  switch (arguments.length) {
    case 1: // rectangle
      bounds = arguments[0];
      break;
    case 2: // location and extent
      bounds = arguments[0].extent(arguments[1]);
      break;
    case 4: // x,y,width, height
      bounds = new Rectangle(arguments[0], arguments[1],
                             arguments[2], arguments[3]);
      break;
    default:
      throw new Error("bad arguments " + arguments);
  }
  return new lively.morphic.Box(bounds);
},
```

Were we to use this method with a solver that understands about array-like objects (such as the one in Babelsberg/R), we allow the solver to make up arguments. Consider the following constraint:

[5] These examples are part of a published artifact [5], and are also available from the Babelsberg/JS repository at `https://github.com/babelsberg/babelsberg-js` (retrieved Feb 25, 2015). For our adaption, we used the versions from the repository.

```
always: {
  myRectangleMorph.equals(Morph.makeRectangle($world.bounds()))
}
```

Here, the desire is to make the world bounds equal to the bounds of the `myRectangleMorph`. However, the solver may decide to modify the arguments array and simply pass four integers corresponding to the bounds of `myRectangleMorph`. The weak stays on the arguments array may not suffice here if other constraints are involved. For example, given a required stay that the world bounds stay at some constant, this should, by extension, keep the morph at the same constant. However, if the solver is free to modify the arguments array, it may just substitute the `$world.bounds` argument.

Although such behavior is surprising, it is very common to check arguments in such a manner in the LivelyKernel and other JavaScript codebases. A future improvement to our design that may increase compatibility may allow such statements that simply do argument checking, without adding them to the constraint. For now, although it is not semantically clean, the practical implementation allows such tests on arguments, but special-cases them so they appear read-only to the solver.

Branching Dispatching based on the number of arguments is a special case of a more general pattern we have encountered. Some methods return one of two expressions, depending on a test. Our semantics does not allow branching in constraints. Such a method encountered frequently is `getPosition`:

```
function getPosition() {
 if (!this.hasFixedPosition() || !this.world())
  return this.morphicGetter('Position');
 else
  return this.world().getScrollOffset().
     addPt(this.morphicGetter('Position'));
}
```

If we allowed such methods in constraints, the solver could change the branch condition, which makes it harder to anticipate how a constraint will behave. We think our semantics adds clarity here by requiring the developer to move the test outside of the method and adding the constraint only for the branch that is chosen.

Benign Side Effects Lazy initialization and caching are used in the constraints in some of the example applications. Prior work has explicitly allowed such benign side effects [4, 5], but our formal semantics disallows them. (We believe that this is an appropriate simplification to the formal semantics, preferring to leave this issue to clear but less formal guidance to programmers and language implementors. However, we are planning to revisit this issue in future work if it proves to be too restrictive.) For example, the LivelyKernel method `Morph.getBounds` is used (often indirectly) in many of the examples. The code below was adapted to focus on the caching – the method also uses branches and local variables, which are also disallowed now:

```
function getBounds() {
 if (this.cachedBounds && !this.hasFixedPosition())
  return this.cachedBounds;
 // ... other code paths guarded with branches
 return this.cachedBounds = this.innerBounds();
}
```

A workaround that we use here is to call `innerBounds` directly, and circumvent the caching. For Morphs for which all child Morphs are contained within the bounds of their parent, this returns the same result.

Another example method directly uses lazy initialization in example code:

```
function getCelsius() {
 if (!this.updater)
  this.updater = setInterval(this.updateCelsiusFromWeb.bind(this), 1000);
 return this.Celsius;
}
```

A workaround here is to move initializers into separate methods and run them before using the field in a constraint.

Local Variables Some methods we encountered declared local variables to split up calculations or name constants. For such methods, manual inlining of the split up calculation into the return expression was required. The original Babelsberg design explicitly allows such methods, as long as the variables are used in static-single-assignment (SSA) fashion [4]:

```
function pressure() {
  var gasConstantDryAir = 287.058, // J/(kg*K)
    density = 1.293, // kg/m^3
    entropyPerVol = gasConstantDryAir * density; // J/K/m^3
  return entropyPerVol * this.K / 1000;
}
```

Edit Constraints Edit constraints [7] support the efficient updating of a solution to the current constraints as a set of input values changes, for example, as a point in a diagram is dragged with the mouse. Edit constraints are supported in the JavaScript and Ruby implementations of Babelsberg, but are not yet accounted for in our formal semantics. Adding them to the semantics should be straightforward.

Finally, our semantics does not include support for edit constraints [7, 8], so existing programs which use them do not currently work. Although incremental solving through edit constraints provides performance benefits for interactive applications, many of the interactive example programs in Babelsberg/JS seem not to use them, or only use them in such a limited fashion that refactoring to ordinary constraints was easily possible. However, performance does suffer in the cases where we removed edit constraints. A future goal for this semantics is thus to include support for edit constraints that can be implemented in a performant manner.

Overall, we think these changes, although they represent additional work for the programmer, improve the clarity of the code and make the interactions between the object-oriented part and the constraints more comprehensible. A future goal will be to see which of these workarounds can be removed, while still maintaining a clean and comprehensible set of rules for the interaction of the object-oriented core and constraints.

References

[1] Nikolaj Björner and Anh-Dung Phan. νZ–Maximal Satisfaction with Z3. In *6th International Symposium on Symbolic Computation in Software Science*, 2014. Invited paper.

[2] Alan Borning. Architectures for cooperating constraint solvers. Technical Report VPRI Memo M-2012-003, Viewpoints Research Institute, Glendale, California, 2012.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer, 2008.

[4] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. Specifying and Solving Constraints on Object Behavior. *Journal of Object Technology*, 13(4):1:1–38, 2014.

[5] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/JS: A Browser-Based Implementation of an Object Constraint Language. In *Proceedings of the 2014 European Conference on Object-Oriented Programming*, pages 411–436. Springer, 2014.

[6] Tim Felgentreff, Todd Millstein, and Alan Borning. Developing a formal semantics for babelsberg: A step-by-step approach. Technical Report TR2014002, Viewpoints Research Institute, Los Angeles, California, 2014.

[7] Bjorn Freeman-Benson and John Maloney. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Proceedings of the 8th Annual IEEE Phoenix Conference on Computers and Communications*, pages 538–542. IEEE, 1989.

[8] Bjorn N Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[9] Dan Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. The Lively Kernel – A Self-Supporting System on a Web Page. In *Proceedings of the Workshop on Self-Sustaining Systems (S3)*, pages 31–50. Springer, 2008.

[10] Mikael Pettersson. RML—A new Language and Implementation for Natural Semantics. In *Programming Language Implementation and Logic Programming*, pages 117–131. Springer, 1994.

[11] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124. ACM, 2013.

This appendix presents the entirety of the semantic rules of Babelsberg/Objects. It also includes the complete source code to the RML semantics, and the source code of our test generation framework.

A. Formalism

Statement	`s`	`::=`	`skip` | `L := e` | `always C` | `once C` | `s;s`
			| `if e then s else s` | `while e do s`
Constraint	`C`	`::=`	ρ `e` | `C` $\wedge$ `C`
Expression	`e`	`::=`	`v` | `L` | `e` $\oplus$ `e`
			| `e.l(`e_1`,...,`e_n`)` | `o` | `new o` | `D`
Identity	`I`	`::=`	`e == e`
Object Literal	`o`	`::=`	`{`l_1`:`e_1`,...,`l_n`:`e_n`}`
L-Value	`L`	`::=`	`x` | `e.l`
Constant	`c`	`::=`	`true` | `false` | `nil` | base type constants
Variable	`x`	`::=`	variable names
Label	`l`	`::=`	record label names
Reference	`r`	`::=`	references to heap records
Dereference	`D`	`::=`	`H(e)`
Method Body	`b`	`::=`	`s; return e` | `return e`
Value	`v`	`::=`	`c` | `r` | `{`l_1`:`v_1`,...,`l_n`:`v_n`}`

$\boxed{\textit{lookup}(v,l) = (x_1 \cdots x_n, b)}$

"Lookup of method `l` in the object or value `v` returns the formal parameter names x_1 through x_n and the method body `b`"

$\boxed{\textit{enter}(E,S,H,C,\ I,v,x_1 \cdots x_n,e_1 \cdots e_n) = (E',S_m,H',C',I')}$

"Invoking a method on `v` with argument names x_1 through x_n and arguments e_1 through e_n constructs the method scope S_m and may update the heap and constraint stores."

$$\frac{\begin{array}{c} <E|S|H|C|I|e_1> \Downarrow <E_1|H_1|C_1|I_1|v_1> \\ \cdots \\ <E_{n-1}|S|H_{n-1}|C_{n-1}|I_{n-1}|e_n> \Downarrow <E_n|H_n|C_n|I_n|v_n> \\ <E_n|S_m|H_n|C_n|I_n|\texttt{self := } v> \longrightarrow <E_0|S_0|H_n|C_n|I_n> \\ <E_0|S_0|H_n|C_n|I_n|x_1 \texttt{ := } v_1> \longrightarrow <E_{n+1}|S_1|H_n|C_n|I_n> \\ \cdots \\ <E_{2n-1}|S_{n-1}|H_n|C_n|I_n|x_n \texttt{ := } v_n> \longrightarrow <E_{2n}|S_n|H_n|C_n|I_n> \end{array}}{\textit{enter}(E,S,H,C,\ I,v,x_1 \cdots x_n,e_1 \cdots e_n) = (E_{2n},S_n,H_n,C_n,I_n)} \text{(ENTER)}$$

$\boxed{<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|v>}$

"Evaluating expression e produces the value v, while possibly having side-effects on everything but the local environment."

$$<E|S|H|C|I|c> \Downarrow <E|H|C|I|c> \qquad (\textsc{E-Const})$$

$$\frac{S(x) = x_g \qquad E(x_g) = v}{<E|S|H|C|I|x> \Downarrow <E|H|C|I|v>} \qquad (\textsc{E-Var})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|r> \qquad H'(r) = \{l_1{:}v_1,\ldots,l_n{:}v_n\}\\ 1 \leqslant i \leqslant n\end{array}}{<E|S|H|C|I|e.l_i> \Downarrow <E'|H'|C'|I'|v_i>} \qquad (\textsc{E-Field})$$

$$\frac{<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|\{l_1{:}v_1,\ldots,l_n{:}v_n\}> \qquad 1 \leqslant i \leqslant n}{<E|S|H|C|I|e.l_i> \Downarrow <E'|H'|C'|I'|v_i>} \qquad (\textsc{E-ValueField})$$

$$<E|S|H|C|I|r> \Downarrow <E|H|C|I|r> \qquad (\textsc{E-Ref})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e_1> \Downarrow <E_0|H_0|C_0|I_0|v_1>\\ <E_0|S|H_0|C_0|I_0|e_2> \Downarrow <E'|H'|C'|I'|v_2>\\ v_1 \ [\![\oplus]\!] \ v_2 = v\end{array}}{<E|S|H|C|I|e_1 \ \oplus \ e_2> \Downarrow <E'|H'|C'|I'|v>} \qquad (\textsc{E-Op})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e_1> \Downarrow <E_0|H_0|C_0|I_0|v>\\ <E|S|H_0|C_0|I_0|e_2> \Downarrow <S'|H'|C'|I'|v>\end{array}}{<E|S|H|C|I|e_1 \ \texttt{==} \ e_2> \Downarrow <E'|H'|C'|I'|\texttt{true}>} \qquad (\textsc{E-IdentityTrue})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e_1> \Downarrow <E_0|H_0|C_0|I_0|v_1>\\ <E|S|H_0|C_0|I_0|e_2> \Downarrow <S'|H'|C'|I'|v_2>\\ v_1 \neq v_2\end{array}}{<E|S|H|C|I|e_1 \ \texttt{==} \ e_2> \Downarrow <E'|H'|C'|I'|\texttt{false}>} \qquad (\textsc{E-IdentityFalse})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E_0|H_0|C_0|I_0|v>\\ \textit{lookup}(v,l) = (x_1 \cdots x_n, s\texttt{; return } e)\\ \textit{enter}(E_0,S,H_0,C_0,\ I_0,v,x_1 \cdots x_n,e_1 \cdots e_n) = (E_1,S_m,H_1,C_1,I_1)\\ <E_1|S_m|H_1|C_1|I_1|s> \longrightarrow <E'|S'|H'|C'|I'>\\ <E'|S'|H'|C'|I'|e> \Downarrow <E''|H''|C''|I''|v_r>\end{array}}{<E|S|H|C|I|e.l(e_1,\ldots,e_n)> \Downarrow <E''|H''|C''|I''|v_r>} \qquad (\textsc{E-Call})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E_0|H_0|C_0|I_0|v>\\ \textit{lookup}(v,l) = (x_1 \cdots x_n,\texttt{return } e)\\ \textit{enter}(E_0,S,H_0,C_0,\ I_0,v,x_1 \cdots x_n,e_1 \cdots e_n) = (E_1,S_m,H_1,C_1,I_1)\\ <E_1|S_m|H_1|C_1|I_1|e> \Downarrow <E'|H'|C'|I'|v_r>\end{array}}{<E|S|H|C|I|e.l(e_1,\ldots,e_n)> \Downarrow <E'|H'|C'|I'|v_r>} \qquad (\textsc{E-CallSimple})$$

$$\frac{\begin{array}{c}\texttt{<E|S|H|C|I|}e_1\texttt{>} \Downarrow \texttt{<}E_1\texttt{|}H_1\texttt{|}C_1\texttt{|}I_1\texttt{|}v_1\texttt{>} \\ \cdots \\ \texttt{<}E_{n-1}\texttt{|S|}H_{n-1}\texttt{|}C_{n-1}\texttt{|}I_{n-1}\texttt{|}e_n\texttt{>} \Downarrow \texttt{<}E_n\texttt{|}H_n\texttt{|}C_n\texttt{|}I_n\texttt{|}v_n\texttt{>} \\ H_n(r)\uparrow \qquad H' = (H_n \bigcup \{(r, \{l_1:v_1,\ldots,l_n:v_n\})\})\end{array}}{\texttt{<E|S|H|C|I|new } \{l_1:e_1,\ldots,l_n:e_n\}\texttt{>} \Downarrow \texttt{<}E_n\texttt{|}H'\texttt{|}C_n\texttt{|}I_n\texttt{|r>}} \quad \text{(E-New)}$$

$$\frac{\begin{array}{c}\texttt{<E|S|H|C|I|}e_1\texttt{>} \Downarrow \texttt{<}E_1\texttt{|}H_1\texttt{|}C_1\texttt{|}I_1\texttt{|}v_1\texttt{>} \\ \cdots \\ \texttt{<}E_{n-1}\texttt{|S|}H_{n-1}\texttt{|}C_{n-1}\texttt{|}I_{n-1}\texttt{|}e_n\texttt{>} \Downarrow \texttt{<}E_n\texttt{|}H_n\texttt{|}C_n\texttt{|}I_n\texttt{|}v_n\texttt{>}\end{array}}{\texttt{<E|S|H|C|I|}\{l_1:e_1,\ldots,l_n:e_n\}\texttt{>} \Downarrow \texttt{<}E_n\texttt{|}H_n\texttt{|}C_n\texttt{|}I_n\texttt{|}\{l_1:v_1,\ldots,l_n:v_n\}\texttt{>}}$$

(E-Value)

$\boxed{E;H \vdash e : T}$

"Expression e has type T under E and H"

$\boxed{E;H \vdash C}$

"Constraint C is well-formed under E and H"

Type $T ::= \texttt{PrimitiveType} \mid \{l_1:T_1,\ldots,l_n:T_n\}$

$$E;H \vdash c : \texttt{PrimitiveType} \quad \text{(T-Constant)}$$

$$\frac{E(x) = v \qquad E;H \vdash v : T}{E;H \vdash x : T} \quad \text{(T-Variable)}$$

$$\frac{E;H \vdash e : \{l_1:T_1,\ldots,l_n:T_n\} \qquad 1 \leqslant i \leqslant n}{E;H \vdash e.l_i : T_i} \quad \text{(T-Field)}$$

$$\frac{H(r)=o \qquad E;H \vdash o : T}{E;H \vdash r : T} \quad \text{(T-Ref)}$$

$$\frac{E;H \vdash e : T}{E;H \vdash H(e) : T} \quad \text{(T-Deref)}$$

$$\frac{E;H \vdash e_1 : \texttt{PrimitiveType} \qquad E;H \vdash e_2 : \texttt{PrimitiveType}}{E;H \vdash e_1 \oplus e_2 : \texttt{PrimitiveType}} \quad \text{(T-Op)}$$

$$\frac{E;H \vdash e_1 : T_1 \cdots E;H \vdash e_n : T_n}{E;H \vdash \{l_1:e_1,\ldots,l_n:e_n\} : \{l_1:T_1,\ldots,l_n:T_n\}} \quad \text{(T-ValueObject)}$$

$$\frac{E;H \vdash e : T}{E;H \vdash \rho \ e} \quad \text{(T-Constraint)}$$

$$\frac{E;H \vdash C_1 : T \qquad E;H \vdash C_2 : T}{E;H \vdash C_1 \wedge C_2} \quad \text{(T-CompoundConstraint)}$$

$\boxed{\mathtt{E;H} \models \mathtt{C}}$

"A call to the solver to solve C yields E and H"

$\boxed{\text{stay}(\mathtt{E}) = \mathtt{C}}$

"Adding weak stays to all values in E yields C"

$\boxed{\text{stay}(\mathtt{H}) = \mathtt{C}}$

"Adding weak stays to all objects on the heap H yields C"

$\boxed{\text{i-stay}(\mathtt{e}) = \mathtt{C}}$

"Creating stays for the expression e yields weak stays for all primitive types and irecords in e and required stays for all references in e"

$\boxed{\text{i-stay}(\mathtt{E}) = \mathtt{C}}$

"Adding stays to all values in E yields weak stays for all primitive types and irecords and required stays for all references"

$\boxed{\text{i-stay}(\mathtt{H}) = \mathtt{C}}$

"Adding stays to all objects on the heap H yields weak stays for all primitive fields and irecords in those objects and required stays for all references"

$$\text{stay}(\emptyset) = \mathtt{true} \qquad \text{(StayEmpty)}$$

$$\frac{\mathtt{E(x)} = \mathtt{v} \qquad \mathtt{E_0} = \mathtt{E}\backslash\{(\mathtt{x},\mathtt{v})\} \qquad \text{stay}(\mathtt{E_0}) = \mathtt{C_0} \qquad \mathtt{C{=}C_0} \wedge \mathtt{weak\ x{=}v}}{\text{stay}(\mathtt{E}) = \mathtt{C}} \qquad \text{(StayOne)}$$

$$\frac{\begin{array}{c}\mathtt{H(r)} = \mathtt{o} \qquad \mathtt{H_0} = \mathtt{H}\backslash\{(\mathtt{r},\mathtt{o})\} \qquad \text{stay}(\mathtt{H_0}) = \mathtt{C_0} \qquad \mathtt{o} = \{\mathtt{l_0{:}v_0,\ldots,l_n{:}v_n}\} \\ C_w = \text{weak } x_{r0} = v_0 \wedge \ldots \wedge \text{weak } x_{rn} = v_n\end{array}}{\text{stay}(\mathtt{H}) = \mathtt{C_0} \ \wedge\ \mathtt{weak\ H(r)\ =\ \{l_0{:}x_{r0},\ldots,l_n{:}x_{rn}\}} \ \wedge\ \mathtt{C_w}} \qquad \text{(StayHeap)}$$

$$\text{i-stay}(\emptyset) = \mathtt{true} \qquad \text{(IStayEmpty)}$$

$$\text{i-stay}(\mathtt{x{=}c}) = \mathtt{weak\ x{=}c} \qquad \text{(IStayConst)}$$

$$\frac{\text{i-stay}(\mathtt{x_{x0}{=}v_0}) = \mathtt{C_{x0}} \cdots \text{i-stay}(\mathtt{x_{xn}{=}v_n}) = \mathtt{C_{xn}} \qquad \mathtt{C_x} = \mathtt{C_{x0}} \wedge \ldots \wedge \mathtt{C_{xn}}}{\text{i-stay}(\mathtt{x{=}\{l_0{:}v_0,\ldots,l_n{:}v_n\}}) = \ \mathtt{required\ x{=}\{l_0{:}x_{x0},\ldots,l_n{:}x_{xn}\}} \ \wedge\ \mathtt{C_x}} \qquad \text{(IStayObject)}$$

$$\text{i-stay}(\mathtt{x{=}r}) = \mathtt{required\ x{=}r} \qquad \text{(IStayRef)}$$

$$\frac{\texttt{E(x)} = \texttt{v} \qquad \texttt{E}_0 = \texttt{E}\backslash\{(\texttt{x},\texttt{v})\} \qquad \text{i-stay}(\texttt{E}_0) = \texttt{C}_0 \qquad \text{i-stay}(\texttt{x=v}) = \texttt{C}_s}{\text{i-stay}(\texttt{E}) = \texttt{C}_0 \ \wedge \ \texttt{C}_s}$$

(IStayOne)

$$\frac{\begin{array}{c}\texttt{H(r)} = \texttt{o} \qquad \texttt{H}_0 = \texttt{H}\backslash\{(\texttt{r},\texttt{o})\} \qquad \text{i-stay}(\texttt{H}_0) = \texttt{C}_0 \qquad \texttt{o} = \{\texttt{l}_0\texttt{:v}_0,\ldots,\texttt{l}_n\texttt{:v}_n\} \\ \text{i-stay}(\texttt{x}_{r0}\texttt{=v}_0) = \texttt{C}_{r0} \cdots \text{i-stay}(\texttt{x}_{rn}\texttt{=v}_n) = \texttt{C}_{rn}\end{array}}{\text{i-stay}(\texttt{H}) = \texttt{C}_0 \ \wedge \ \texttt{required H(r) = }\{\texttt{l}_0\texttt{:x}_{r0},\ldots,\texttt{l}_n\texttt{:x}_{rn}\} \ \wedge \ \texttt{C}_{r0} \ \wedge \ \ldots \ \wedge \ \texttt{C}_{rn}}$$

(IStayHeap)

$$\boxed{\texttt{<E,S,H,C,I,e>} \rightsquigarrow \texttt{<E}',\texttt{e}_C\texttt{,e}'\texttt{>}}$$

"Inlining expression e from the local environment S turns into expression e′. To connect variables across method calls, the constraint expression e_C is returned."

$$\texttt{<E,S,H,C,I,c>} \rightsquigarrow \texttt{<E,true,c>} \qquad \text{(I-Const)}$$

$$\frac{\texttt{S(x) = x}_g}{\texttt{<E,S,H,C,I,x>} \rightsquigarrow \texttt{<E,true,x}_g\texttt{>}} \qquad \text{(I-Var)}$$

$$\frac{\texttt{<E,S,H,C,I,e}_1\texttt{>} \rightsquigarrow \texttt{<E}_1\texttt{,e}_{C_1}\texttt{,e}_1'\texttt{>} \cdots \texttt{<E,S,H}_{n-1}\texttt{,C,I,e}_n\texttt{>} \rightsquigarrow \texttt{<E}_n\texttt{,e}_{C_n}\texttt{,e}_n'\texttt{>}}{\texttt{<E,S,H,C,I,}\{\texttt{l}_1\texttt{:e}_1,\ldots,\texttt{l}_n\texttt{:e}_n\}\texttt{>} \rightsquigarrow \texttt{<E}_n\texttt{,e}_{C_1}\wedge\cdots\wedge\texttt{e}_{C_n}\texttt{,}\{\texttt{l}_1\texttt{:e}_1',\ldots,\texttt{l}_n\texttt{:e}_n'\}\texttt{>}}$$

(I-Value)

$$\frac{\texttt{<E,S,H,C,I,e>} \rightsquigarrow \texttt{<E}',\texttt{e}_C\texttt{,e}'\texttt{>} \qquad \texttt{<E}'\texttt{|S|H|C|I|e>} \Downarrow \texttt{<E}''\texttt{|H|C|I|r>}}{\texttt{<E,S,H,C,I,e.l>} \rightsquigarrow \texttt{<E}',\texttt{e}_C\wedge\texttt{e}'\texttt{=r,H(e}'\texttt{).l>}}$$

(I-Field)

$$\frac{\begin{array}{c}\texttt{<E,S,H,C,I,e>} \rightsquigarrow \texttt{<E}',\texttt{e}_C\texttt{,e}'\texttt{>} \\ \texttt{<E}'\texttt{|S|H|C|I|e>} \Downarrow \texttt{<E}''\texttt{|H|C|I|}\{\texttt{l}_1\texttt{:v}_1,\ldots,\texttt{l}_n\texttt{:v}_n\}\texttt{>}\end{array}}{\texttt{<E,S,H,C,I,e.l>} \rightsquigarrow \texttt{<E}',\texttt{e}_C\texttt{,e}'\texttt{.l>}} \qquad \text{(I-ValueField)}$$

$$\texttt{<E,S,H,C,I,r>} \rightsquigarrow \texttt{<E,true,r>} \qquad \text{(I-Ref)}$$

$$\frac{\texttt{<E,S,H,C,I,e}_1\texttt{>} \rightsquigarrow \texttt{<E}',\texttt{e}_{C_a}\texttt{,e}_a\texttt{>} \qquad \texttt{<E}'\texttt{,S,H,C,I,e}_2\texttt{>} \rightsquigarrow \texttt{<E}''\texttt{,e}_{C_b}\texttt{,e}_b\texttt{>}}{\texttt{<E,S,H,C,I,e}_1 \oplus \texttt{e}_2\texttt{>} \rightsquigarrow \texttt{<E}''\texttt{,e}_{C_a}\wedge\texttt{e}_{C_b}\texttt{,e}_a \oplus \texttt{e}_b\texttt{>}}$$

(I-Op)

$$\frac{\texttt{<E,S,H,C,I,e}_1\texttt{>} \rightsquigarrow \texttt{<E}',\texttt{e}_{C_a}\texttt{,e}_a\texttt{>} \qquad \texttt{<E}'\texttt{,S,H,C,I,e}_2\texttt{>} \rightsquigarrow \texttt{<E}''\texttt{,e}_{C_b}\texttt{,e}_b\texttt{>}}{\texttt{<E,S,H,C,I,e}_1\texttt{ == e}_2\texttt{>} \rightsquigarrow \texttt{<E}''\texttt{,e}_{C_a}\wedge\texttt{e}_{C_b}\texttt{,e}_a\texttt{ == e}_b\texttt{>}}$$

(I-Identity)

$$\frac{\begin{array}{c}\texttt{<E|S|H|C|I|e>} \Downarrow \texttt{<E}'\texttt{|H|C|I|v>} \\ \textit{lookup}(\texttt{v,l}) = (\texttt{x}_1 \cdots \texttt{x}_n\texttt{,s; return e}) \\ \textit{enter}(\texttt{E}',\texttt{S,H,C, I,v,x}_1 \cdots \texttt{x}_n\texttt{,e}_1 \cdots \texttt{e}_n) = (\texttt{E}'',\texttt{S}_m\texttt{,H,C,I}) \\ \texttt{<E}''\texttt{|S}_m\texttt{|H|C|I|s>} \longrightarrow \texttt{<E}'''\texttt{|S}'\texttt{|H|C|I >} \\ \texttt{<E}'''\texttt{|S}'\texttt{|H|C|I|e>} \Downarrow \texttt{<E}''''\texttt{|H|C|I|v}_r\texttt{>}\end{array}}{\texttt{<E,S,H,C,I,e.l(e}_1,\ldots,\texttt{e}_n\texttt{)>} \rightsquigarrow \texttt{<E}''''\texttt{,true,v}_r\texttt{>}} \qquad \text{(I-Call)}$$

$$\frac{\begin{array}{c}\texttt{<E,S,H,C,I,e}_0\texttt{>} \leadsto \texttt{<E}',\texttt{e}_{C_0},\texttt{e}_0'\texttt{>}\\ \texttt{<E}'\texttt{|S|H|C|I|e}_0\texttt{>} \Downarrow \texttt{<E}''\texttt{|H|C|I|v>}\\ \textit{lookup}(\texttt{v},\texttt{l}) = (\texttt{x}_1 \cdots \texttt{x}_n\texttt{,return e})\\ \textit{enter}(\texttt{E}'',\texttt{S,H,C, I,v,x}_1 \cdots \texttt{x}_n,\texttt{e}_1 \cdots \texttt{e}_n) = (\texttt{E}''',\texttt{S}_m,\texttt{H,C,I})\\ \texttt{<E}''',\texttt{S,H,C,I,e}_1\texttt{>} \leadsto \texttt{<E}_1,\texttt{e}_{C_1},\texttt{e}_1'\texttt{>} \cdots \texttt{<E}_{n-1},\texttt{S,H,C,I,e}_n\texttt{>} \leadsto \texttt{<E}_n,\texttt{e}_{C_n},\texttt{e}_n'\texttt{>}\\ \texttt{S}_m(\texttt{self}) = \texttt{x}_{g_{\texttt{self}}} \qquad \texttt{S}_m(\texttt{x}_1) = \texttt{x}_{g_1} \cdots \texttt{S}_m(\texttt{x}_n) = \texttt{x}_{g_n}\\ \texttt{e}_C = (\texttt{x}_{g_{\texttt{self}}}\texttt{=e}_0' \wedge \texttt{x}_{g_1}\texttt{=e}_1' \wedge \cdots \wedge \texttt{x}_{g_n}\texttt{=e}_n')\\ \texttt{<E}_n,\texttt{S}_m,\texttt{H,C,I,e>} \leadsto \texttt{<E}_n',\texttt{e}_{C_m},\texttt{e}'\texttt{>}\end{array}}{\texttt{<E,S,H,C,I,e}_0\texttt{.l(e}_1,\ldots,\texttt{e}_n\texttt{)>} \leadsto \texttt{<E}_n',\texttt{e}_C \wedge \texttt{e}_{C_m} \wedge \texttt{e}_{C_0} \wedge \texttt{e}_{C_1} \wedge \cdots \wedge \texttt{e}_{C_n},\texttt{e}'\texttt{>}}$$

(I-MultiWayCall)

$\boxed{\texttt{<E,H,I,C >} \leadsto \texttt{<E}',\texttt{C>}}$

"Re-inlining the constraint store `C` returns a constraint `C`"

$\boxed{\texttt{<E,H,C,I >} \leadsto \texttt{<E}',\texttt{C>}}$

"Re-inlining the constraint store `I` returns a constraint `C`"

$$\texttt{<E,H,I,}\emptyset\texttt{>} \leadsto \texttt{<E,true>} \qquad \text{(I-ReinlineEmptyC)}$$

$$\texttt{<E,H,C,}\emptyset\texttt{>} \leadsto \texttt{<E,true>} \qquad \text{(I-ReinlineEmptyI)}$$

$$\frac{\begin{array}{c}\texttt{C}_0 = \texttt{C} \setminus \{(\texttt{S}, \rho\, \texttt{e})\} \qquad \texttt{<E,H,I,C}_0\texttt{>} \leadsto \texttt{<E}_0,\texttt{C}_0\texttt{>}\\ \texttt{<E}_0,\texttt{S,H,C}_0,\texttt{I,e>} \leadsto \texttt{<E}',\texttt{e}_{C_e},\texttt{e}'\texttt{>}\end{array}}{\texttt{<E,H,I,C >} \leadsto \texttt{<E}',\texttt{C}_0 \wedge \rho\ (\texttt{e}' \wedge \texttt{e}_{C_e})\texttt{>}} \quad \text{(I-ReinlineC)}$$

$$\frac{\begin{array}{c}\texttt{I}_0 = \texttt{I} \setminus \{(\texttt{S,required e})\} \qquad \texttt{<E,H,C,I}_0\texttt{>} \leadsto \texttt{<E}_0,\texttt{C}_0\texttt{>}\\ \texttt{<E}_0,\texttt{S,H,C,I}_0,\texttt{e>} \leadsto \texttt{<E}',\texttt{e}_{C_e},\texttt{e}'\texttt{>}\end{array}}{\texttt{<E,H,C,I >} \leadsto \texttt{<E}',\texttt{C}_0 \wedge \texttt{required}\ (\texttt{e}' \wedge \texttt{e}_{C_e})\texttt{>}} \quad \text{(I-ReinlineI)}$$

$\boxed{\texttt{<E|H|C|I|I|C>} \Longrightarrow \texttt{<E}'\texttt{|H}'\texttt{>}}$

"Solving `I` and `C` first for identities and then values yields `E′` and `H′`"

$$\frac{\begin{array}{c}\text{stay}(\texttt{E}) = \texttt{C}_{\texttt{E}_s} \qquad \text{stay}(\texttt{H}) = \texttt{C}_{\texttt{H}_s} \qquad \texttt{<E,H,C,I >} \leadsto \texttt{<E}_i,\texttt{C}_i\texttt{>}\\ \texttt{E}';\texttt{H}' \models (\texttt{C}_i \wedge \texttt{C}_{\texttt{E}_s} \wedge \texttt{C}_0 \wedge \texttt{C}_{\texttt{H}_s} \wedge \texttt{e}_1\texttt{=e}_2)\\ \text{i-stay}(\texttt{E}') = \texttt{C}_{\texttt{E}'_s} \qquad \text{i-stay}(\texttt{H}') = \texttt{C}_{\texttt{H}'_s} \qquad \texttt{<E}',\texttt{H}',\texttt{I,C >} \leadsto \texttt{<E}_c,\texttt{C>}\\ \texttt{E}_c;\texttt{H}' \vdash \texttt{C} \qquad \texttt{E}'';\texttt{H}'' \models (\texttt{C} \wedge \texttt{C}_0 \wedge \texttt{C}_{\texttt{E}'_s} \wedge \texttt{C}_{\texttt{H}'_s} \wedge \texttt{e}_1\texttt{=e}_2)\end{array}}{\texttt{<E|H|C|I|e}_1\texttt{ == e}_2\texttt{|C}_0\texttt{>} \Longrightarrow \texttt{<E}''\texttt{|H}''\texttt{>}}$$

(TwoPhaseUpdate)

$\boxed{\texttt{<E|S|H|C|I|s>} \longrightarrow \texttt{<E}'\texttt{|S}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}}$

"Execution starting from configuration $<E|E|H|C|I|s>$ ends in configuration $<E'|E'|H'|C'|I'>$."

$$<E|S|H|C|I|\texttt{skip}> \longrightarrow <E|S|H|C|I\ > \qquad (\textsc{S-Skip})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|s_1> \longrightarrow <E'|S'|H'|C'|I'> \\ <E'|S'|H'|C'|I'|s_2> \longrightarrow <E''|S''|H''|C''|I''>\end{array}}{<E|S|H|C|I|s_1;s_2> \longrightarrow <E''|S''|H''|C''|I''>} \;(\textsc{S-Seq})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|\texttt{true}> \\ <E'|S|H'|C'|I'|s_1> \longrightarrow <E''|S'|H''|C''|I''>\end{array}}{<E|S|H|C|I|\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2> \longrightarrow <E''|S'|H''|C''|I''>} \;(\textsc{S-IfThen})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|\texttt{false}> \\ <E'|S|H'|C'|I'|s_2> \longrightarrow <E''|S'|H''|C''|I''>\end{array}}{<E|S|H|C|I|\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2> \longrightarrow <E''|S'|H''|C''|I''>} \;(\textsc{S-IfElse})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|\texttt{true}> \\ <E'|S|H'|C'|I'|s> \longrightarrow <E''|S'|H''|C''|I''> \\ <E''|S'|H''|C''|I''|\texttt{while } e \texttt{ do } s> \longrightarrow <E'''|S''|H'''|C'''|I'''>\end{array}}{<E|S|H|C|I|\texttt{while } e \texttt{ do } s> \longrightarrow <E'''|S''|H'''|C'''|I'''>} \;(\textsc{S-WhileDo})$$

$$\frac{<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|\texttt{false}>}{<E|S|H|C|I|\texttt{while } e \texttt{ do } s> \longrightarrow <E'|S|H'|C'|I'>} \;(\textsc{S-WhileSkip})$$

$$\frac{\begin{array}{c}S(x)\uparrow \qquad E(x_g)\uparrow \qquad S' = S \bigcup \{(x, x_g)\} \\ <E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|v> \\ <E'|H'|C'|I'|x_g\texttt{==}v|\texttt{true}> \Longrightarrow <E''|H''>\end{array}}{<E|S|H|C|I|x \texttt{ := } e> \longrightarrow <E''|S'|H''|C'|I'>} \;(\textsc{S-AsgnNewLocal})$$

$$\frac{\begin{array}{c}S(x) = x_g \\ <E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|v> \\ <E'|H'|C'|I'|x_g\texttt{==}v|\texttt{true}> \Longrightarrow <E''|H''>\end{array}}{<E|S|H|C|I|x \texttt{ := } e> \longrightarrow <E''|S|H''|C|I\ >} \;(\textsc{S-AsgnLocal})$$

$$\frac{\begin{array}{c}<E|S|H|C|I|e> \Downarrow <E'|H'|C'|I'|v> \\ <E',S,H',C',I',e_l.l> \rightsquigarrow <E'',e_c,e'> \\ <E''|H'|C'|I'|e'\texttt{==}v|e_c> \Longrightarrow <E'''|H''>\end{array}}{<E|S|H|C|I|e_l.l \texttt{ := } e> \longrightarrow <E'''|S|H''|C'|I'>} \;(\textsc{S-AsgnLValue})$$

$$\frac{\begin{array}{c}C_0 = \rho\ e \qquad <E,S,H,C,I,e> \rightsquigarrow <E',e_{C_e},e'> \qquad C_0' = \rho\ (e' \wedge e_{C_e}) \\ E';H \vdash C_0' \qquad \text{i-stay}(E') = C_{E_s} \qquad \text{i-stay}(H) = C_{H_s} \\ <E',H,I,C\ > \rightsquigarrow <E'',C> \qquad E''';H' \models (C \wedge C_{E_s} \wedge C_{H_s} \wedge C_0')\end{array}}{<E|S|H|C|I|\texttt{once } C_0> \longrightarrow <E'''|S|H'|C|I\ >} \;(\textsc{S-Once})$$

$$\frac{\begin{array}{c}\texttt{<E|S|H|C|I|once } C_0\texttt{>} \longrightarrow \texttt{<}E'\texttt{|S|}H'\texttt{|C|I >}\\ C' = C \bigcup \{(S, C_0)\}\end{array}}{\texttt{<E|S|H|C|I|always } C_0\texttt{>} \longrightarrow \texttt{<}E'\texttt{|S|}H'\texttt{|}C'\texttt{|I >}} \quad \text{(S-ALWAYS)}$$

$$\frac{\begin{array}{cc}\texttt{<E|S|H|C|I|}e_0\texttt{>} \Downarrow \texttt{<}E_0\texttt{|H|C|I|v>} & \texttt{<}E_0\texttt{|S|H|C|I|}e_1\texttt{>} \Downarrow \texttt{<}E_1\texttt{|H|C|I|v>}\\ \texttt{<}E_1\texttt{,S,H,C,I,}e_0\texttt{>} \rightsquigarrow \texttt{<}E_2\texttt{,}e_{C_0}\texttt{,}e_0'\texttt{>} & \texttt{<}E_2\texttt{,S,H,C,I,}e_1\texttt{>} \rightsquigarrow \texttt{<}E_3\texttt{,}e_{C_1}\texttt{,}e_1'\texttt{>}\end{array}}{1\ \texttt{<E|S|H|C|I|once } e_0 \texttt{ == } e_1\texttt{>} \longrightarrow \texttt{<}E_3\texttt{|S|H|C|I >}}$$
(S-ONCEIDENTITY)

$$\frac{\begin{array}{c}\texttt{<E|S|H|C|I|once } e_0 \texttt{ == } e_1\texttt{>} \longrightarrow \texttt{<}E'\texttt{|S|}H'\texttt{|C|I >}\\ I' = I \bigcup \{(S, e_0 = e_1)\}\end{array}}{\texttt{<E|S|H|C|I|always } e_0 \texttt{ == } e_1\texttt{>} \longrightarrow \texttt{<}E'\texttt{|S|}H'\texttt{|C|}I'\texttt{>}} \quad \text{(S-ALWAYSIDENTITY)}$$

B. Executable Semantics

To build the executable semantics we used the RML mirror at `https://github.com/timfel/rml`. The central file of an RML project is the language definition including syntax and semantic rules, which in our case is `babelsberg.rml` given below. RML translates such files into C code which can then be compiled, creating a header file with function declarations for the parser and main file. The entry point is a C file written by hand that parses the source using Yacc. The Yacc source includes the generated header file to create the RML AST nodes during parsing. A basic layout of such a project, including lexer/parser, main.c, and a Makefile is part of the RML distribution, and our executable semantics is based on that. Our basic project structure is:

```
/
├── z3 (the solver binary)
├── rml
│   └── ... (source tree of RML)
├── babelsberg-objects
│   ├── main.c
│   ├── parser.y
│   ├── lexer.l
│   ├── babelsberg.rml
│   ├── helper.rml
│   ├── printer.rml
│   ├── solver.rml
│   ├── solver.c
│   ├── cisolver.rb
│   ├── Makefile
│   └── examples
│       ├── 1.txt (example program 1)
│       ├── 1.env (expected solutions to solver calls in 1)
│       ├── 2.txt (example program 2)
│       ├── 2.illegal (marker file that typing should fail in 2)
│       └── ... (more example programs)
└── javascript-test-generator
    ├── main.c
    ├── javascript.rml
    ├── assertions.rml
    ├── assertions.c
    ├── find_example.rb
    ├── assert.rb
    ├── scaffold.js
    └── Makefile
```

We present here the source code of the lexer and parser, then the code of the `babelsberg.rml` file that implements our semantic rules (omitting the printing and helper rules, as these are straightforward), and finally, we present the interface to the solver given in `solver.c`.

B.1. Lexer

The lexer rules are mostly straightforward. The source code for the lexer includes `babelsberg.h` and `rml.h` to generate the correct boxes for literals immediately in the tokenization phase.

```
%{
#define YYSTYPE void*
#include "parser.h"
#include "rml.h"
#include "babelsberg.h"

int lex_icon(void);
int lex_ident(void);
int lex_rcon(void);

%}

whitespace [ \t\n]+
letter [a-zA-Z_]
lowerletter [a-z]
upperletter [A-Z]
questionmark "?"
ident {lowerletter}({letter}|{digit})*{questionmark}?
global {upperletter}({letter}|{digit})*
digit [0-9]
digits {digit}+
pt "."
quote "\""
sign [-]
exponent ([eE]{sign}?{digits})
rcon1 {sign}?{digits}({pt}{digits}?)?{exponent}
rcon2 {sign}?{digits}?{pt}{digits}{exponent}?
rcon {rcon1}|{rcon2}
icon1 {digits}
icon2 {sign}{digits}
icon {icon1}|{icon2}
string {quote}({letter}|{digit}|{whitespace})+{quote}
amp "#"
reference {amp}{digits}

%%

{whitespace} ;
{string} return lex_string();
{icon} return lex_rcon(); /* convert ints to T_REALCONST */
{rcon} return lex_rcon(); /* T_REALCONST */
{reference} return lex_ref();
"H" return T_H_DEREF;
"new" return T_NEW;
":=" return T_ASSIGN;
"+" return T_ADD;
"-" return T_SUB;
"*" return T_MUL;
"/" return T_DIV;
"==" return T_IDENTICAL;
"<" return T_LESSTHAN;
"<=" return T_LEQUAL;
"!=" return T_NEQUAL;
```

```
"=" return T_EQUAL;
">=" return T_GEQUAL;
">" return T_GREATERTHAN;
"&&" return T_AND;
"and" return T_AND;
"or" return T_OR;
"(" return T_LPAREN;
")" return T_RPAREN;
"{" return T_LBRACE;
"}" return T_RBRACE;
":" return T_COLON;
"," return T_COMMA;
{pt} return T_DOT;
";" return T_SEMIC;
"skip" return T_SKIP;
"always" return T_ALWAYS;
"once" return T_ONCE;
"weak" return T_WEAK;
"medium" return T_MEDIUM;
"required" return T_REQUIRED;
"if" return T_IF;
"then" return T_THEN;
"else" return T_ELSE;
"while" return T_WHILE;
"do" return T_DO;
"true" return T_TRUE;
"false" return T_FALSE;
"nil" return T_NIL;
{ident} return lex_ident(); /* T_IDENT */
{global} return lex_global(); /* T_GLOBAL */

%%

int lex_rcon(void)
{
  yylval= (void*)mk_rcon(atof(yytext));
  return T_REALCONST;
}

int lex_ident(void)
{
  yylval = (void*)mk_scon(yytext);
  return T_IDENT;
}

int lex_global(void)
{
  yylval = (void*)mk_scon(yytext);
  return T_GLOBAL;
}

int lex_string(void)
{
  yylval = (void*)mk_scon(yytext);
  return T_STRING;
}

int lex_ref(void)
{
  yylval = (void*)mk_icon(atoi(yytext + 1));
  return T_REF;
}
```

B.2. Parser

The Yacc parser uses the constructors for AST boxes that RML generated from our abstract syntax definition in `babelsberg.rml` to create the abstract syntax tree. This tree is thus immediately in a form that can be processed by RML rules.

```
%{
#include <stdio.h>
#include "rml.h"
#include "babelsberg.h"

#define YYSTYPE void*
extern void* absyntree;

/* int yydebug=1; */

%}

%token T_SEMIC
%token T_ASSIGN
%token T_IDENT
%token T_GLOBAL
%token T_REALCONST
%token T_STRING
%token T_H_DEREF
%token T_REF
%token T_NEW
%token T_IDENTICAL
%token T_LPAREN T_RPAREN
%token T_LESSTHAN
%token T_LEQUAL
%token T_EQUAL
%token T_NEQUAL
%token T_GEQUAL
%token T_GREATERTHAN
%token T_OR
%token T_AND
%token T_ADD
%token T_SUB
%token T_MUL
%token T_DIV

%token T_LBRACE
%token T_RBRACE
%token T_COLON
%token T_COMMA
%token T_DOT

%token T_GARBAGE

%token T_SKIP
%token T_ALWAYS
%token T_ONCE
%token T_WEAK
%token T_MEDIUM
%token T_REQUIRED
%token T_IF
%token T_THEN
%token T_ELSE
%token T_WHILE
%token T_DO
%token T_TRUE
%token T_FALSE
```

```
%token T_NIL


%token T_ERR

%left T_OR
%left T_AND
%left T_LESSTHAN T_LEQUAL T_EQUAL T_NEQUAL T_GEQUAL T_GREATERTHAN
%left T_ADD T_SUB
%left T_MUL T_DIV

%%

/* Yacc BNF grammar of the expression language BabelsbergPs */

program : statement
                  { absyntree = babelsberg__PROGRAM($1);}

statement : T_SKIP
                  { $$ = babelsberg__SKIP; }
             | lvalue T_ASSIGN expression
                  { $$ = babelsberg__ASSIGN($1, $3); }
             | T_ALWAYS constraint
                  { $$ = babelsberg__ALWAYS($2); }
             | T_ONCE constraint
                  { $$ = babelsberg__ONCE($2); }
             | statement T_SEMIC statement
                  { $$ = babelsberg__SEQ($1, $3); }
             | T_IF expression T_THEN statement T_ELSE statement
                  { $$ = babelsberg__IF($2, $4, $6); }
             | T_WHILE expression T_DO statement
                  { $$ = babelsberg__WHILE($2, $4); }

constraint : rho expression
                  { $$ = babelsberg__CONSTRAINT($1, $2); }
             | expression
                  { $$ = babelsberg__CONSTRAINT(babelsberg__REQUIRED, $1); }
             | constraint T_AND constraint
                  { $$ = babelsberg__COMPOUNDCONSTRAINT($1, $3); }

rho : T_WEAK
                  { $$ = babelsberg__WEAK; }
             | T_MEDIUM
                  { $$ = babelsberg__MEDIUM; }
             | T_REQUIRED
                  { $$ = babelsberg__REQUIRED; }

expression : value
                  { $$ = babelsberg__VALUE($1); }
             | lvalue
                  { $$ = babelsberg__LVALUE($1); }
             | expression woperation expression %prec T_MUL
                  { $$ = babelsberg__OP($1, $2, $3); }
             | expression soperation expression %prec T_ADD
                  { $$ = babelsberg__OP($1, $2, $3); }
             | expression comparison expression %prec T_EQUAL
                  { $$ = babelsberg__OP($1, $2, $3); }
             | expression combination expression %prec T_AND
                  { $$ = babelsberg__OP($1, $2, $3); }
             | expression disjunction expression %prec T_OR
                  { $$ = babelsberg__OP($1, $2, $3); }
             | expression T_IDENTICAL expression %prec T_EQUAL
                  { $$ = babelsberg__IDENTITY($1, $3); }
             | T_GLOBAL T_LPAREN callargs T_RPAREN
```

```
                  { $$ = babelsberg__CALL(babelsberg__VALUE(babelsberg__K(babelsberg__NIL)),
                        $1, $3); }
            | expression T_DOT label T_LPAREN callargs T_RPAREN
                  { $$ = babelsberg__CALL($1, $3, $5); }
            | T_LBRACE objectliteral T_RBRACE
                  { $$ = babelsberg__IRECORD($2); }
            | T_NEW T_LBRACE objectliteral T_RBRACE
                  { $$ = babelsberg__UIDRECORD($3); }
            | dereference
                  { $$ = babelsberg__DEREF($1); }

callargs : /* empty */
                 { $$ = mk_nil(); }
            | expression
                 { $$ = mk_cons($1, mk_nil()); }
            | expression T_COMMA callargs
                 { $$ = mk_cons($1, $3); }

objectliteral : /* empty */
                 { $$ = mk_nil(); }
            | fieldexpression
                 { $$ = mk_cons($1, mk_nil()); }
            | fieldexpression T_COMMA objectliteral
                 { $$ = mk_cons($1, $3); }

fieldexpression : label T_COLON expression
                 { $$ = mk_box2(1, $1, $3); }

lvalue : variable
                  { $$ = babelsberg__VARIABLE($1); }
            | expression T_DOT label
                  { $$ = babelsberg__FIELD($1, $3); }
            | dereference  /* only for solver model parsing */
                  { $$ = babelsberg__ASSIGNDEREF($1); }

constant : T_TRUE
                  { $$ = babelsberg__TRUE; }
            | T_FALSE
                  { $$ = babelsberg__FALSE; }
            | T_NIL
                  { $$ = babelsberg__NIL; }
            | T_REALCONST
                  { $$ = babelsberg__REAL($1);}
            | T_STRING
                  { $$ = babelsberg__STRING($1);}

variable : T_IDENT
                  { $$ = $1; }

label : T_IDENT
                  { $$ = $1; }

reference : T_REF
                  { $$ = $1; }

dereference : T_H_DEREF T_LPAREN expression T_RPAREN
                  { $$ = $3; }

value : constant
                  { $$ = babelsberg__K($1); }
            /*
             * | objectliteral
             * { $$ = babelsberg__O($1); }
             */
            | reference
                 { $$ = babelsberg__R($1); }

soperation : T_ADD
                 { $$ = babelsberg__ADD;}
            | T_SUB
                 { $$ = babelsberg__SUB;}
woperation : T_MUL
                 { $$ = babelsberg__MUL;}
            | T_DIV
                 { $$ = babelsberg__DIV;}

comparison : T_LESSTHAN
                 { $$ = babelsberg__LESSTHAN;}
            | T_LEQUAL
                 { $$ = babelsberg__LEQUAL;}
            | T_EQUAL
                 { $$ = babelsberg__EQUAL;}
            | T_NEQUAL
                 { $$ = babelsberg__NEQUAL;}
            | T_GEQUAL
                 { $$ = babelsberg__GEQUAL;}
            | T_GREATERTHAN
                 { $$ = babelsberg__GEQUAL;}

combination : T_AND
                 { $$ = babelsberg__AND;}

disjunction : T_OR
                 { $$ = babelsberg__OR;}
```

B.3. RML Semantics

The next listings combined form the file `babelsberg.rml`. The listing is split according to the semantic rules that are implemented for readability.

B.3.1. Abstract Syntax

We presented the abstract syntax in Section 2.1. Note that during compilation, RML generates a header file with constructor functions for all AST nodes that we define, which we used above in our lexer and parser.

```
module babelsberg:
  datatype Program = PROGRAM of Statement

  datatype Statement = SKIP
                | ASSIGN of LValue * Exp
                | ALWAYS of Constraint
                | ONCE of Constraint
                | SEQ of Statement * Statement
                | IF of Exp * Statement * Statement
                | WHILE of Exp * Statement

  datatype Constraint = CONSTRAINT of Rho * Exp
                 | COMPOUNDCONSTRAINT of Constraint * Constraint

  datatype Rho = WEAK | MEDIUM | REQUIRED

  datatype Exp = VALUE of Value
           | LVALUE of LValue
           | OP of Exp * Op * Exp
```

```
              | IDENTITY of Exp * Exp
              | CALL of Exp * Label * Exp list
              | IRECORD of ObjectLiteral
              | UIDRECORD of ObjectLiteral
              | DEREF of Dereference

  type ObjectLiteral = (Label * Exp) list

  datatype LValue = VARIABLE of Variable | FIELD of Exp * Label
                | ASSIGNDEREF of Dereference (* just for the parsing *)

  datatype Constant = TRUE | FALSE | NIL | REAL of real | STRING of string

  type Variable = string

  type Label = string

  type Reference = int

  type Dereference = Exp

  datatype MethodBody = METHOD of Statement * Exp | SIMPLE of Exp

  datatype Value = K of Constant | O of ObjectLiteral | R of Reference

(* Helper types *)
  datatype Op = ADD | SUB | MUL | DIV
               | LESSTHAN | LEQUAL | EQUAL | NEQUAL | GEQUAL | GREATERTHAN
               | AND | OR

(* Type syntax *)
  datatype Type = PRIMITIVE | TRECORD of (Label * Type) list

(* Bindings and environments *)
  type Env = (Variable * Value) list
  type Scope = (Variable * Variable) list
  type Heap = (Reference * ObjectLiteral) list
  type Cstore = (Scope * Constraint) list
  type Istore = (Scope * Constraint) list

  relation evalprogram: Program => ()
  relation eval: (Env, Scope, Heap, Cstore, Istore, Exp) => (Env, Heap, Cstore, Istore,
       Value)

  relation alert: (string list, Exp list, string list) => ()
end
```

B.3.2. Main Entry Point

The next relation is the main entry point for evaluation. It receives the entire AST from the parser and evaluates it to the end or until it gets stuck.

```
relation evalprogram: Program => () =
  rule print "starting to evaluate\n" &
      step([], [], [], [], [], statement) => (E,S,H,C,I)
      ------------------------------------------------
      evalprogram(PROGRAM(statement))
end
```

B.3.3. Method Lookup

$\boxed{\textit{lookup}(v,l) = (x_1 \cdots x_n, b)}$

The lookup relation has no rules in the formal semantics. For our executable semantics, we define all methods that we use in the test cases inline, that is, we provide rules for all method names that we use in programs and return an argument list and a method body.

```
relation lookup: (Value, Label) => (Variable list, MethodBody) =
 (* hard coded methods *)
 axiom lookup(_, "one") => ([], (* return 1.0 *)
          SIMPLE(VALUE(K(REAL(1.0)))))

 axiom lookup(_, "double") => ([], (* return 2*self *)
          SIMPLE(OP(VALUE(K(REAL(2.0))),MUL,LVALUE(VARIABLE("self")))))

 axiom lookup(_, "Require_min_balance") => (["acct","min"], (* always acct.balance > min *)
          METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                        OP(LVALUE(FIELD(LVALUE(VARIABLE("acct")),"balance")),
                            GREATERTHAN,
                            LVALUE(VARIABLE("min?"))))),
              LVALUE(VARIABLE("self"))))

 axiom lookup(_, "Has_min_balance") => (["acct","min"], (* return acct.balance > min *)
          SIMPLE(OP(LVALUE(FIELD(LVALUE(VARIABLE("acct")),"balance")),
                GREATERTHAN,
                LVALUE(VARIABLE("min")))))

 (* {x: arg1, y: arg2} *)
 axiom lookup(_, "Point") => (["x", "y"],
             SIMPLE(IRECORD([("x", LVALUE(VARIABLE("x"))),
                    ("y", LVALUE(VARIABLE("y")))])))

 (* (self.upper_left.addPt(self.lower_right).divPtScalar(2) *)
 axiom lookup(_, "center") => ([],
             SIMPLE(CALL(CALL(LVALUE(FIELD(LVALUE(VARIABLE("self")), "upper_left")),
                      "addPt",
                      [LVALUE(FIELD(LVALUE(VARIABLE("self")), "lower_right"))]),
                  "divPtScalar",
                  [VALUE(K(REAL(2.0)))])))

 (* {x: self.x + pt.x, y: self.y + pt.y} *)
 axiom lookup(_, "addPt") => (["pt"],
             SIMPLE(CALL(VALUE(K(NIL)),
                  "Point",
                  [OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"x")),
                    ADD,
                    LVALUE(FIELD(LVALUE(VARIABLE("pt")),"x"))),
                   OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"y")),
                    ADD,
                    LVALUE(FIELD(LVALUE(VARIABLE("pt")),"y")))])))

 (* {x: self.x / scale, y: self.y / scale} *)
 axiom lookup(_, "divPtScalar") => (["scale"],
                SIMPLE(CALL(VALUE(K(NIL)),
                      "Point",
                      [OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"x")),
                        DIV,
                        LVALUE(VARIABLE("scale"))),
                      OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"y")),
                        DIV,
                        LVALUE(VARIABLE("scale")))])))
```

```

(* self.x == other.x && self.y == other.y *)
axiom lookup(_, "ptEq") => (["other"],
               SIMPLE(OP(OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"x")),
                            EQUAL,
                            LVALUE(FIELD(LVALUE(VARIABLE("other")),"x"))),
                        AND,
                        OP(LVALUE(FIELD(LVALUE(VARIABLE("self")),"y")),
                            EQUAL,
                            LVALUE(FIELD(LVALUE(VARIABLE("other")),"y"))))))

(* always medium i = 5; return i + 1 *)
axiom lookup(_, "Test") => (["i"],
              METHOD(ALWAYS(CONSTRAINT(MEDIUM,
                  OP(LVALUE(VARIABLE("i")),EQUAL,VALUE(K(REAL(5.0)))))),
                      OP(LVALUE(VARIABLE("i")),ADD,VALUE(K(REAL(1.0))))))

axiom lookup(_, "MutablePointNew") => (["x", "y"],
                    SIMPLE(UIDRECORD([("x", LVALUE(VARIABLE("x"))),
                             ("y", LVALUE(VARIABLE("y")))])))

axiom lookup(_, "WindowNew") => ([],
                 SIMPLE(UIDRECORD([("window", VALUE(K(TRUE)))])))

axiom lookup(_, "CircleNew") => ([],
                 SIMPLE(UIDRECORD([("circle", VALUE(K(TRUE)))])))


axiom lookup(_, "Makeeq") => (["x","y"],
               METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                          IDENTITY(LVALUE(VARIABLE("x")),LVALUE(VARIABLE("y"))))),
                   LVALUE(VARIABLE("self"))))

axiom lookup(_, "MakeIdentical") => (["a","b"],
               METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                          IDENTITY(LVALUE(VARIABLE("a")),LVALUE(VARIABLE("b"))))),
                   LVALUE(VARIABLE("self"))))

axiom lookup(_, "Testalwaysxequal5") => (["x"],
               METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                     OP(LVALUE(VARIABLE("x")),EQUAL,
                         VALUE(K(REAL(5.0)))))),
                   LVALUE(VARIABLE("x"))))

axiom lookup(_, "Testalwaysaequalsbplus3") => (["a", "b"],
               METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                     OP(LVALUE(VARIABLE("a")),EQUAL,
                         OP(LVALUE(VARIABLE("b")),ADD,VALUE(K(REAL(3.0))))))),
                   LVALUE(VARIABLE("a"))))

axiom lookup(_, "Testpointxequals5") => (["p"],
               METHOD(ALWAYS(CONSTRAINT(REQUIRED,
                     OP(LVALUE(FIELD(LVALUE(VARIABLE("p")),"x")),EQUAL,
                         VALUE(K(REAL(5.0)))))),
                   LVALUE(VARIABLE("p"))))

axiom lookup(_, "TestXGetsXPlus3ReturnX") => (["x"],
               METHOD(ASSIGN(VARIABLE("x"),OP(LVALUE(VARIABLE("x")),ADD,VALUE(K(REAL(3.0))))),
                   LVALUE(VARIABLE("x"))))

end
```

B.3.4. Method Activation

$\boxed{\textit{enter}(E,S,H,C,\ I,v,x_1 \cdots x_n,e_1 \cdots e_n) = (E',S_m,H',C',I')}$

This rule implements method activation, by evaluating each argument expression and creating a new method scope with the argument names bound to the results of the argument evaluation. It uses helper relations to evaluate and assign variables in a new scope.

```
relation enter: (Env, Scope, Heap, Cstore, Istore, Value, Variable list, Exp list) \
                => (Env, Scope, Heap, Cstore, Istore) =
  rule evalEach(E, S, H, C, I, argexps) => (En, Hn, Cn, In, argvals) &
       assignEachFresh(En, [], Hn, Cn, In, "self" :: argnames, v :: argvals) \
                   => (E2n, Sn, Hn, Cn, In)
       ----------------------------------------------------------------------
       enter(E, S, H, C, I, v, argnames, argexps) => (E2n, Sn, Hn, Cn, In)
end

relation evalEach: (Env, Scope, Heap, Cstore, Istore, Exp list) => (Env, Heap, Cstore,
     Istore, Value list) =
  axiom evalEach(E,S,H,C,I,[]) => (E,H,C,I,[])

  rule eval(E,S,H,C,I,e) => (E',H',C',I',v) &
      evalEach(E',S,H',C',I',rest) => (E'',H'',C'',I'',values)
      ----------------------------------------------------------
      evalEach(E,S,H,C,I,e :: rest) => (E'',H'',C'',I'',v :: values)
end

relation assignEachFresh: (Env, Scope, Heap, Cstore, Istore, Variable list, Value list) => (
     Env, Scope, Heap, Cstore, Istore) =
  axiom assignEachFresh(E,S,H,C,I,[],[]) => (E,S,H,C,I)

  rule tick() => i & int_string(i) => is & string_append(x,is) => xg &
      list_append(S,[(x,xg)]) => S' &
      list_append(E,[(xg,v)]) => E' &
      assignEachFresh(E',S',H,C,I,xrest, vrest) => (E'',S'',H,C,I)
      -------------------------------------------------------------------
      assignEachFresh(E,S,H,C,I,x :: xrest, v :: vrest) => (E'',S'',H,C,I)
end
```

B.3.5. Expression Evaluation

$\boxed{\texttt{<E|S|H|C|I|e>} \Downarrow \texttt{<E'|H'|C'|I'|v>}}$

Here we implement the rules for expression evaluation. Besides the preconditions from the formal semantics, we include debug statements that print which formal rule we have matched.

```
relation eval: (Env, Scope, Heap, Cstore, Istore, Exp) \
           => (Env, Heap, Cstore, Istore, Value) =

  axiom eval(E,S,H,C,I,VALUE(K(c))) => (E,H,C,I,K(c))

  rule alert(["E-Var(", x],[],[")\n"]) &
      Util.lookupScope(S, x) => xg & Util.lookupEnv(E, xg) => v
      ------------------------------------------------------
      eval(E,S,H,C,I, LVALUE(VARIABLE(x))) => (E,H,C,I,v)

  rule eval(E,S,H,C,I,e) => (E',H',C',I',R(r)) &
      alert(["E-Field(", l, ")\n"], [], []) &
```

```
    Util.lookupHeap(H', r) => fvalues &
    Util.lookupRecord(fvalues, l) => v
    --------------------------------------------------
    eval(E,S,H,C,I, LVALUE(FIELD(e, l))) => (E',H',C',I',v)

rule eval(E,S,H,C,I,e) => (E',H',C',I',O(fvalues)) &
    alert(["E-ValueField(", l, ")"], [], []) &
    Util.lookupRecord(fvalues, l) => v
    --------------------------------------------------
    eval(E,S,H,C,I, LVALUE(FIELD(e, l))) => (E',H',C',I',v)

axiom eval(E,S,H,C,I, VALUE(R(r))) => (E,H,C,I, R(r))

rule eval(E,S,H,C,I,e1) => (E',H',C',I',v1) &
    Util.should_short_circuit(op, v1) => (true, v) &
    print "E-Op (short circuit)\n"
    ----------------------------------------------
    eval(E,S,H,C,I, OP(e1, op, e2)) => (E',H',C',I',v)

rule alert(["E-Op\n"],[],[]) &
    eval(E,S,H,C,I,e1) => (E',H',C',I',v1) &
    eval(E',S,H',C',I',e2) => (E'',H'',C'',I'',v2) &
    Util.apply_binop(op,v1,v2) => v
    -------------------------------------------------
    eval(E,S,H,C,I, OP(e1, op, e2)) => (E'',H'',C'',I'',v)

rule eval(E,S,H,C,I,e1) => (E',H',C',I',v1) &
    eval(E',S,H',C',I',e2) => (E'',H'',C'',I'',v2) &
    v1 = v2 &
    alert(["E-IdentityTrue\n"],[],[])
    --------------------------------------------------------
    eval(E,S,H,C,I, IDENTITY(e1, e2)) => (E'',H'',C'',I'',K(TRUE))

rule eval(E,S,H,C,I,e1) => (E',H',C',I',v1) &
    eval(E',S,H',C',I',e2) => (E'',H'',C'',I'',v2) &
    not v1 = v2 &
    alert(["E-IdentityFalse\n"],[],[])
    ---------------------------------------------------------
    eval(E,S,H,C,I, IDENTITY(e1, e2)) => (E'',H'',C'',I'',K(FALSE))

rule eval(E,S,H,C,I,e) => (E0,H0,C0,I0,v) &
    lookup(v,l) => (argnames, METHOD(s, returne)) &
    alert(["E-Call\n"],[],[]) &
    enter(E0,S,H0,C0,I0,v,argnames,argexps) => (E1,Sm,H1,C1,I1) &
    step(E1,Sm,H1,C1,I1,s) => (E',S',H',C',I') &
    eval(E',S',H',C',I',returne) => (E'',H'',C'',I'',vr)
    -------------------------------------------------------
    eval(E,S,H,C,I,CALL(e,l,argexps)) => (E'',H'',C'',I'',vr)

rule alert(["E-CallSimple\n"],[],[]) &
    eval(E,S,H,C,I,e) => (E0,H0,C0,I0,v) &
    lookup(v,l) => (argnames, SIMPLE(returne)) &
    enter(E0,S,H0,C0,I0,v,argnames,argexps) => (E1,Sm,H1,C1,I1) &
    eval(E1,Sm,H1,C1,I1,returne) => (E',H',C',I',vr)
    -------------------------------------------------------
    eval(E,S,H,C,I,CALL(e,l,argexps)) => (E',H',C',I',vr)

rule alert(["E-New\n"],[],[]) &
    evalEachField(E,S,H,C,I,fieldexps) => (En,Hn,Cn,In,fieldvalues) &
    tick() => r &
    list_append([(r, fieldvalues)], Hn) => H'
    -----------------------------------------------------------
    eval(E,S,H,C,I,UIDRECORD(fieldexps)) => (En,H',Cn,In,R(r))
```

```
  rule alert(["E-Value\n"],[],[]) &
      evalEachField(E,S,H,C,I,fieldexps) => (En,Hn,Cn,In,fieldvalues)
      ------------------------------------------------------------------
      eval(E,S,H,C,I,IRECORD(fieldexps)) => (En,Hn,Cn,In,O(fieldvalues))
end

relation evalEachField: (Env, Scope, Heap, Cstore, Istore, ObjectLiteral) \
                  => (Env, Heap, Cstore, Istore, ObjectLiteral) =
  axiom evalEachField(E,S,H,C,I,[]) => (E,H,C,I,[])

  rule eval(E,S,H,C,I,e) => (E',H',C',I',v) &
      evalEachField(E',S,H',C',I',rest) => (E'',H'',C'',I'',values)
      ---------------------------------------------------------------------------------
      evalEachField(E,S,H,C,I,(l,e) :: rest) => (E'',H'',C'',I'',(l,VALUE(v)) :: values)
end
```

B.3.6. Structural Type Checking

$\boxed{E;H \vdash e : T}$

The next rules implement our structural type checking rules for expressions. Again, we add debug statements to print which rule we have matched.

```
relation tC: (Env, Heap, Exp) => Type =
  axiom tC(E, H, VALUE(K(c))) => PRIMITIVE
  rule alert(["T-Var\n"],[],[]) &
      Util.lookupEnv(E, x) => v &
      tC(E,H,VALUE(v)) => T
      ---------------------------------
      tC(E,H, LVALUE(VARIABLE(x))) => T

  rule alert(["T-Field\n"],[],[]) &
     tC(E,H,e) => TRECORD(ftypes) &
     Util.lookupRecordType(ftypes, l) => T
     -------------------------------------
     tC(E,H, LVALUE(FIELD(e, l))) => T

  rule alert(["T-Ref\n"],[],[]) &
      Util.lookupHeap(H, r) => fvalues &
      tC(E,H,VALUE(O(fvalues))) => T
      ----------------------------------
      tC(E,H,VALUE(R(r))) => T

  rule alert(["T-Op\n"],[],[]) &
      tC(E,H,e1) => PRIMITIVE &
      tC(E,H,e2) => PRIMITIVE
      ------------------------------------
      tC(E,H, OP(e1, op, e2)) => PRIMITIVE

  rule alert(["T-ValueObject\n"],[],[]) &
     tCFields(E,H, fvalues) => ftypes
     ---------------------------------------------
     tC(E,H, VALUE(O(fvalues))) => TRECORD(ftypes)

  rule alert(["T-Deref\n"],[],[]) &
     tC(E,H, e) => T
     ----------------------------
     tC(E,H, DEREF(e)) => T
end

relation tCFields: (Env, Heap, ObjectLiteral) => (Label * Type) list =
  axiom tCFields(E,H,[]) => []
```

```

 rule tC(E,H,e) => T &
     tCFields(E,H,rest) => ftypes
     -----------------------------------------------
     tCFields(E,H, (l,e) :: rest) => ((l,T) :: ftypes)
end
```

$\boxed{E;H \vdash C}$

For this judgment we deviated from the naming scheme of the formalism.

```
relation welltyped: (Env, Heap, Constraint) => () =
 rule tC(E,H,e) => T
     ---------------------------------
     welltyped(E,H, CONSTRAINT(rho, e))

 rule welltyped(E,H,C1) & welltyped(E,H,C2)
     -----------------------------------------
     welltyped(E,H, COMPOUNDCONSTRAINT(C1, C2))
end
```

B.3.7. Constraint Solving

$\boxed{E;H \models C}$

This judgment has no rules in the formalism. In our formal semantics, we use printing rules to convert all expressions into Z3's SMT2 syntax, as described in Section 2.3.1. We then call the solver and parse the solution using another helper relation, `Print.parseEnvironment`.

```
with "solver.rml"
with "printer.rml"
with "helper.rml"

relation models: Constraint => (Env, Heap) =
 rule Print.printC(C) => plainCs &
     Print.pRefDom(C) => refDom &
     Print.pLabDom(C) => labDom &
     Print.pDefs(C) => defs &
     Print.printZ3C(C) => Cs &
     string_append_list(["(set-option :pp.decimal true)(set-option :model.compact true)\n",
                     refDom, "\n", labDom, "\n
; Next block of declarations are the same everywhere
; We use a Union type for values
(declare-datatypes () ((Value (Record (rec (Array Label Real)))
                      (Real (real Real))
                      (Bool (bool Bool))
                      (Reference (ref Reference)))))
; A default record has 'invalid' for all fields
(declare-const iRec (Array Label (Value)))
(assert (= iRec ((as const (Array Label (Value))) (Reference invalid))))
(declare-const invalidR Real) (assert (= invalidR 1334))
(declare-const vRec (Array Label (Real)))
(assert (= vRec ((as const (Array Label (Real))) invalidR)))
; Records are (Array Label (Value))
(declare-fun H ((Value)) (Array Label (Value)))
(assert (and (= (H (Reference invalid)) iRec) (= (H (Reference nil)) iRec)))

(define-fun plus ((x (Value)) (y (Value))) (Value)
            (Real (+ (real x) (real y))))
(define-fun minus ((x (Value)) (y (Value))) (Value)
            (Real (- (real x) (real y))))
(define-fun times ((x (Value)) (y (Value))) (Value)
            (Real (* (real x) (real y))))
(define-fun divide ((x (Value)) (y (Value))) (Value)
            (Real (/ (real x) (real y))))

(define-fun lessthan ((x (Value)) (y (Value))) (Value)
            (Bool (< (real x) (real y))))
(define-fun leq ((x (Value)) (y (Value))) (Value)
            (Bool (<= (real x) (real y))))
(define-fun greaterthan ((x (Value)) (y (Value))) (Value)
            (Bool (> (real x) (real y))))
(define-fun geq ((x (Value)) (y (Value))) (Value)
            (Bool (>= (real x) (real y))))

(define-fun equal ((x (Value)) (y (Value))) (Value)
            (Bool (= x y)))
(define-fun notequal ((x (Value)) (y (Value))) (Value)
            (Bool (not (= x y))))

(define-fun bbband ((x (Value)) (y (Value))) (Value)
            (Bool (and (bool x) (bool y))))
(define-fun bbbor ((x (Value)) (y (Value))) (Value)
            (Bool (or (bool x) (bool y))))\n",
     defs, "\n", Cs, "\n(check-sat)\n(get-model)\n"]) => srr &
     Solver.solve(plainCs, srr) => El &
     print "\n" &
     Print.parseEnvironment(El, [], []) => (E, H) &
     print "\n"
     --------------------------------------------------------
     models(C) => (E, H)
end
```

B.3.8. Stay Constraints

$\boxed{\text{stay}(\mathsf{E}) = \mathsf{C}}$

$\boxed{\text{stay}(\mathsf{H}) = \mathsf{C}}$

$\boxed{\text{i-stay}(e) = \mathsf{C}}$

$\boxed{\text{i-stay}(\mathsf{E}) = \mathsf{C}}$

$\boxed{\text{i-stay}(\mathsf{H}) = \mathsf{C}}$

The next rules generate our stay constraints.

```
relation stay: Env => Constraint =
 axiom stay([]) => CONSTRAINT(REQUIRED, VALUE(K(TRUE)))

 rule alert(["StayOne\n"],[],[]) &
     stay(E0) => C0
     ----------------------------------------------------------------
     stay((x, v) :: E0) => COMPOUNDCONSTRAINT(C0,
         CONSTRAINT(WEAK, OP(LVALUE(VARIABLE(x)), EQUAL, VALUE(v))))
end
```

```

relation stayH: Heap => Constraint =
  axiom stayH([]) => CONSTRAINT(REQUIRED, VALUE(K(TRUE)))

  rule alert(["StayHeap\n"],[],[]) &
      stayH(H0) => C0
      ------------------------------------------------------------------
      stayH((r, o) :: H0) => COMPOUNDCONSTRAINT(C0,
           CONSTRAINT(WEAK, OP(DEREF(VALUE(R(r))), EQUAL, VALUE(O(o)))))
end

relation stayFields: (Reference, ObjectLiteral) => (Constraint, ObjectLiteral) =
  axiom stayFields(_, []) => (CONSTRAINT(REQUIRED, VALUE(K(TRUE))), [])

  rule stayFields(r, rest) => (C, o) &
      int_string(r) => is & string_append_list(["ref", is, "_", l]) => x
      -----------------------------------------------------------------
      stayFields(r, (l, e) :: rest) => (COMPOUNDCONSTRAINT(
              CONSTRAINT(WEAK, OP(LVALUE(VARIABLE(x)), EQUAL, e)), C),
              (l, LVALUE(VARIABLE(x))) :: o)
end

relation identityStay: Env => Constraint =
  axiom identityStay([]) => CONSTRAINT(REQUIRED, VALUE(K(TRUE)))
  rule alert(["StayRef\n"],[],[]) &
      identityStay(E0) => C0
      -----------------------------------------------------------------------
      identityStay((x, R(r)) :: E0) => COMPOUNDCONSTRAINT(C0,
           CONSTRAINT(REQUIRED, OP(LVALUE(VARIABLE(x)), EQUAL, VALUE(R(r)))))

  rule alert(["StayObj\n"],[],[]) & identityStay(E0) => C0 & stay([(x, O(o))]) => C1
      ---------------------------------------------------------------------------
      identityStay((x, O(o)) :: E0) => COMPOUNDCONSTRAINT(C0, C1)

  rule alert(["StayConst\n"],[],[]) & identityStay(E0) => C0 & stay([(x, K(c))]) => C1
      -----------------------------------------------------------------------------
      identityStay((x, K(c)) :: E0) => COMPOUNDCONSTRAINT(C0, C1)
end

relation identityStayH: Heap => Constraint =
  axiom identityStayH([]) => CONSTRAINT(REQUIRED, VALUE(K(TRUE)))

  rule alert(["IdentityStayHeap\n"],[],[]) &
      identityStayH(H0) => C0
      ------------------------------------------------------------------
      identityStayH((r, o) :: H0) => COMPOUNDCONSTRAINT(C0,
           CONSTRAINT(WEAK, OP(DEREF(VALUE(R(r))), EQUAL, VALUE(O(o)))))
end

relation identityStayFields: (Reference, ObjectLiteral) => (Constraint, ObjectLiteral) =
  axiom identityStayFields(_, []) => (CONSTRAINT(REQUIRED, VALUE(K(TRUE))), [])

  rule identityStayFields(r, rest) => (C, o) &
      int_string(r) => is &string_append(is, "_") => is' & string_append(is', l) => x &
      identityStay([(x, v)]) => C0
      ------------------------------------------------------------------------------
      identityStayFields(r, (l, VALUE(v)) :: rest) => (COMPOUNDCONSTRAINT(C0, C),
                                            (l, LVALUE(VARIABLE(x))) :: o)
end
```

B.3.9. Expression Inlining

$\boxed{\texttt{<E,S,H,C,I,e>} \rightsquigarrow \texttt{<E}',\texttt{e}_C,\texttt{e}'\texttt{>}}$

$\boxed{\texttt{<E,H,I,C >} \rightsquigarrow \texttt{<E}',\texttt{C>}}$

$\boxed{\texttt{<E,H,C,I >} \rightsquigarrow \texttt{<E}',\texttt{C>}}$

Below we give the rules for the above judgments. As before, debug prints inform which semantic rules is implemented by which RML rule.

```
relation inline: (Env, Scope, Heap, Cstore, Istore, Exp) => (Env, Exp, Exp) =
  axiom inline(E,S,H,C,I,VALUE(K(c))) => (E,VALUE(K(TRUE)),VALUE(K(c)))

  rule alert(["I-Var"],[],[]) &
      Util.lookupScope(S,x) => xg
      ------------------------------------------------------------------------
      inline(E,S,H,C,I,LVALUE(VARIABLE(x))) => (E,VALUE(K(TRUE)),LVALUE(VARIABLE(xg)))

  rule alert(["I-Record"],[],[]) &
      inlineEachField(E,S,H,C,I,o) => (En,ec,o')
      --------------------------
      inline(E,S,H,C,I,IRECORD(o)) => (En,ec,VALUE(O(o')))

  rule alert(["I-Record"],[],[]) &
      inlineEachField(E,S,H,C,I,o) => (En,ec,o')
      --------------------------
      inline(E,S,H,C,I,VALUE(O(o))) => (En,ec,VALUE(O(o')))

  rule alert(["I-Field"],[],[]) &
      inline(E,S,H,C,I,e) => (E',ec,e') &
      eval(E',S,H,C,I,e) => (E'',H,C,I,R(r))
      ------------------------------------------------------------------------
      inline(E,S,H,C,I,LVALUE(FIELD(e,l))) => (E',OP(ec,AND,OP(e',EQUAL,VALUE(R(r)))),
                                  LVALUE(FIELD(DEREF(e'),l)))

  rule alert(["I-ValueField"],[],[]) &
      inline(E,S,H,C,I,e) => (E',ec,e') &
      eval(E',S,H,C,I,e) => (E'',H,C,I,O(o))
      ------------------------------------------------------------
      inline(E,S,H,C,I,LVALUE(FIELD(e,l))) => (E',ec,LVALUE(FIELD(e',l)))

  axiom inline(E,S,H,C,I,VALUE(R(r))) => (E,VALUE(K(TRUE)),VALUE(R(r)))

  rule alert(["I-Op"],[],[]) &
      inline(E,S,H,C,I,e1) => (E',eca,ea) &
      inline(E',S,H,C,I,e2) => (E'',ecb,eb)
      ------------------------------------------------------------
      inline(E,S,H,C,I,OP(e1,op,e2)) => (E'',OP(eca,AND,ecb),OP(ea,op,eb))

  rule alert(["I-Identity"],[],[]) &
      inline(E,S,H,C,I,e1) => (E',eca,ea) &
      inline(E',S,H,C,I,e2) => (E'',ecb,eb)
      ------------------------------------------------------------------
      inline(E,S,H,C,I,IDENTITY(e1,e2)) => (E'',OP(eca,AND,ecb),IDENTITY(ea,eb))

  rule eval(E,S,H,C,I,e) => (E',H,C,I,v) & lookup(v,l) => (argnames,METHOD(s,e)) &
      alert(["I-Call(", l],[],[")"]) &
      enter(E',S,H,C,I,v,argnames,argexps) => (E'',Sm,H,C,I) &
      step(E'',Sm,H,C,I,s) => (E''',S',H',C',I') &
      H' = H & C' = C & I' = I & (* this is just explicit unification *)
      eval(E''',S',H,C,I,e) => (E'''',H,C,I,vr)
      ---------------------------------------------------------------
      inline(E,S,H,C,I,CALL(e,l,argexps)) => (E'''',VALUE(K(TRUE)),VALUE(vr))

  rule inline(E,S,H,C,I,e0) => (E',ec0,e0') & eval(E',S,H,C,I,e0) => (E'',H,C,I,v) &
      lookup(v,l) => (argnames,SIMPLE(e)) &
```

```
     alert(["I-SimpleCall(", l],[],[")"]) &
     enter(E'',S,H,C,I,v,argnames,argexps) => (E''',Sm,H,C,I) &
     inlineEach(E''',S,H,C,I,argexps) => (En,ecn,inlinedargexps) &
     Util.lookupScope(Sm,"self") => xgself &
     Util.lookupScopeEach(Sm,argnames) => globalargnames &
     Util.pairwiseEqualEach(xgself :: globalargnames,
                     e0' :: inlinedargexps) => argequalities &
     Util.combineEach(argequalities) => ec &
     inline(En,Sm,H,C,I,e) => (En',ecm,e') &
     Util.combineEach([ec0, ec, ecm, ecn]) => eC
     ----------------------------------------------------------------
     inline(E,S,H,C,I,CALL(e0,l,argexps)) => (En',eC,e')
end

relation inlineEach: (Env, Scope, Heap, Cstore, Istore, Exp list) => (Env, Exp, Exp list) =
 axiom inlineEach(E,S,H,C,I,[]) => (E,VALUE(K(TRUE)),[])

 rule inline(E,S,H,C,I,e) => (E',eC,e') &
     inlineEach(E',S,H,C,I,rest) => (E'',restC,rest')
     ---------------------------------------------------------------------
     inlineEach(E,S,H,C,I,e :: rest) => (E'',OP(eC,AND,restC),e' :: rest')
end

relation inlineEachField: (Env, Scope, Heap, Cstore, Istore, ObjectLiteral) \
                   => (Env, Exp, ObjectLiteral) =
 axiom inlineEachField(E,S,H,C,I,[]) => (E,VALUE(K(TRUE)),[])

 rule inline(E,S,H,C,I,e) => (E',eC,e') &
     inlineEachField(E',S,H,C,I,rest) => (E'',restC,rest')
     ----------------------------------------------------------------------------------
     inlineEachField(E,S,H,C,I,(l,e) :: rest) => (E'',OP(eC,AND,restC),(l,e') :: rest')
end

relation reinlineC: (Env, Heap, Istore, Cstore) => (Env, Constraint) =
 axiom reinlineC(E,H,I,[]) => (E,CONSTRAINT(REQUIRED, VALUE(K(TRUE))))

 rule reinlineC(E,H,I,Cstore0) => (E0,C0) &
     inline(E0,S,H,Cstore0,I,e) => (E',eC,e')
     ------------------------------------------------------------------------------
     reinlineC(E,H,I,(S,CONSTRAINT(rho,e)) :: Cstore0) => (E',COMPOUNDCONSTRAINT(C0,
                                             CONSTRAINT(rho,OP(e',AND,eC))))
end

relation reinlineI: (Env, Heap, Cstore, Istore) => (Env, Constraint) =
 axiom reinlineI(E,H,C,[]) => (E,CONSTRAINT(REQUIRED, VALUE(K(TRUE))))

 rule reinlineI(E,H,C,Istore0) => (E0,C0) &
     inline(E0,S,H,C,Istore0,e) => (E',eC,e')
     ------------------------------------------------------------------------------------
     reinlineI(E,H,C,(S,CONSTRAINT(REQUIRED,e)) :: Istore0) => (E',COMPOUNDCONSTRAINT(C0,
                                               CONSTRAINT(REQUIRED,OP(e',AND,eC))))
end
```

B.3.10. Statement Evaluation

$\boxed{\texttt{<E|H|C|I|I|C>} \Longrightarrow \texttt{<E}'\texttt{|H}'\texttt{>}}$

$\boxed{\texttt{<E|S|H|C|I|s>} \longrightarrow \texttt{<E}'\texttt{|S}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}}$

As the final set of rules we implement from the formalism, we give the statement evaluation rules.

```
relation twoPhaseUpdate: (Env, Heap, Cstore, Istore, Exp, Constraint) => (Env, Heap) =
  rule stay(E) => CEs & stayH(H) => CHs & reinlineI(E,H,C,I) => (Ei,Ci) &
       models(COMPOUNDCONSTRAINT(COMPOUNDCONSTRAINT(COMPOUNDCONSTRAINT(Ci,CEs),C0),
              COMPOUNDCONSTRAINT(CHs,CONSTRAINT(REQUIRED,OP(e1,EQUAL,e2))))) => (E',H') &
       identityStay(E') => CEs' & identityStayH(H') => CHs' &
       reinlineC(E',H',I,C) => (Ec,Cc) & welltyped(Ec,H',Cc) &
       models(COMPOUNDCONSTRAINT(COMPOUNDCONSTRAINT(COMPOUNDCONSTRAINT(Cc,C0),
                                        COMPOUNDCONSTRAINT(CEs',CHs')),
                          CONSTRAINT(REQUIRED,OP(e1,EQUAL,e2)))) => (E'',H'')
       ------------------------------------------------------------------------------------------
       twoPhaseUpdate(E,H,C,I,IDENTITY(e1,e2),C0) => (E'',H'')
end

relation step: (Env, Scope, Heap, Cstore, Istore, Statement) \
           => (Env, Scope, Heap, Cstore, Istore) =

  axiom step(E,S,H,C,I,SKIP) => (E,S,H,C,I)

  rule alert(["S-Seq\n"],[],[]) &
       step(E, S, H, C, I, s1) => (E',S',H',C',I') &
       step(E',S',H',C',I',s2) => (E'',S'',H'',C'',I'')
       ---------------------------------------------------
       step(E,S,H,C,I,SEQ(s1,s2)) => (E'',S'',H'',C'',I'')

  rule eval(E,S,H,C,I,e) => (E',H',C',I',K(TRUE)) &
       alert(["S-IfThen\n"],[],[]) &
       step(E',S,H',C',I',s1) => (E'',S',H'',C'',I'')
       -----------------------------------------------------
       step(E,S,H,C,I,IF(e, s1, s2)) => (E'',S',H'',C'',I'')

  rule eval(E,S,H,C,I,e) => (E',H',C',I',v) &
       not v = K(TRUE) &
       alert(["S-IfElse\n"],[],[]) &
       step(E',S,H',C',I',s2) => (E'',S',H'',C'',I'')
       -----------------------------------------------------
       step(E,S,H,C,I,IF(e, s1, s2)) => (E'',S',H'',C'',I'')

  rule eval(E,S,H,C,I,e) => (E',H',C',I',K(TRUE)) &
       alert(["S-WhileDo\n"],[],[]) &
       step(E',S,H',C',I',s) => (E'',S',H'',C'',I'') &
       step(E'',S',H'',C'',I'', WHILE(e, s)) => (E''',S'',H''',C''',I''')
       -----------------------------------------------------------------
       step(E,S,H,C,I,WHILE(e, s)) => (E''',S'',H''',C''',I''')

  rule eval(E,S,H,C,I,e) => (E',H',C',I',v) &
       not v = K(TRUE) &
       alert(["S-WhileSkip\n"],[],[])
       ------------------------------------------------
       step(E,S,H,C,I, WHILE(e, s)) => (E',S,H',C',I')

  rule Util.lookupScope(S,x) => xg & alert(["S-AsgnLocal(",x,")\n"],[],[]) &
       eval(E,S,H,C,I,e) => (E',H',C',I',v) &
       twoPhaseUpdate(E',H',C',I',IDENTITY(LVALUE(VARIABLE(xg)),VALUE(v)),
                 CONSTRAINT(REQUIRED,VALUE(K(TRUE)))) => (E'',H'')
       ---------------------------------------------------------------------
       step(E,S,H,C,I,ASSIGN(VARIABLE(x),e)) => (E'',S,H'',C',I')

  rule not Util.lookupScope(S,x) => _ & alert(["S-AsgnNewLocal(",x,") := "],[e],["\n"]) &
       tick() => i & int_string(i) => istring & string_append(x,istring) => xg &
       list_append(S,[(x,xg)]) => S' &
       eval(E,S,H,C,I,e) => (E',H',C',I',v) &
       twoPhaseUpdate(E',H',C',I',IDENTITY(LVALUE(VARIABLE(xg)),VALUE(v)),
                 CONSTRAINT(REQUIRED,VALUE(K(TRUE)))) => (E'',H'')
       ------------------------------------------------------------------------------------
```

```
      step(E,S,H,C,I,ASSIGN(VARIABLE(x),e)) => (E'',S',H'',C',I')

  rule alert(["S-AsgnLValue\n"],[],[]) &
      eval(E,S,H,C,I,e) => (E',H',C',I',v) &
      inline(E',S,H',C',I',LVALUE(FIELD(el,l))) => (E'',eC,e') &
      twoPhaseUpdate(E'',H',C',I',IDENTITY(e',VALUE(v)),CONSTRAINT(REQUIRED,eC)) => (E''',H
          '')
      ---------------------------------------------------------------------------------------

      step(E,S,H,C,I,ASSIGN(FIELD(el,l),e)) => (E''',S,H'',C',I')

  rule alert(["S-OnceIdentity\n"],[],[]) &
      eval(E,S,H,C,I,e0) => (E0,H,C,I,v0) & eval(E0,S,H,C,I,e1) => (E1,H,C,I,v1) & v0 = v1 &
      inline(E1,S,H,C,I,e0) => (E2,eC0,e0') & inline(E2,S,H,C,I,e1) => (E3,eC1,e1')
      -------------------------------------------------------------------------------------
      step(E,S,H,C,I, ONCE(CONSTRAINT(REQUIRED, IDENTITY(e0, e1)))) => (E3,S,H,C,I)

  rule alert(["S-AlwaysIdentity\n"],[],[]) &
      step(E,S,H,C,I, ONCE(CONSTRAINT(REQUIRED, IDENTITY(e0, e1)))) => (E',S,H,C,I) &
      list_append(I,[(S,CONSTRAINT(REQUIRED, IDENTITY(e0, e1)))]) => I'
      -------------------------------------------------------------------------------
      step(E,S,H,C,I, ALWAYS(CONSTRAINT(REQUIRED, IDENTITY(e0, e1)))) => (E',S,H,C,I')

  rule alert(["S-Once\n"],[],[]) &
      inline(E,S,H,C,I,e) => (E',eC,e') &
      welltyped(E',H, CONSTRAINT(rho,OP(e',AND,eC))) &
      identityStay(E') => CEs & identityStayH(H) => CHs &
      reinlineC(E',H,I,C) => (E'', Cc) &
      models(COMPOUNDCONSTRAINT(COMPOUNDCONSTRAINT(CEs, CHs),
              COMPOUNDCONSTRAINT(Cc, CONSTRAINT(rho,OP(e',AND,eC))))) => (E''',H')
      -----------------------------------------------------------------------------
      step(E,S,H,C,I,ONCE(CONSTRAINT(rho,e))) => (E''',S,H',C,I)

  rule alert(["S-Always\n"],[],[]) &
      step(E,S,H,C,I, ONCE(C0)) => (E',S,H',C,I) &
      list_append(C,[(S,C0)]) => C'
      -----------------------------------------
      step(E,S,H,C,I, ALWAYS(C0)) => (E',S,H',C',I)
end
```

B.3.11. Debugging

The last relation we include corresponds to no rule in the semantics, but is used for debugging. RML does not include conditional compilation, so to deactivate debugging statements, we uncomment the first axiom of this debugging relation, which matches anything and maps it to an empty list, short-circuiting the following debug prints.

```
(* debugging *)
relation alert: (string list, Exp list, string list) => () =
  (* axiom alert(_, _, _) => () *)

  rule string_append_list(s1) => s1' & print s1' &
      Print.printE(e) => es & print es &
      string_append_list(s2) => s2' & print s2'
      -----------------------------------------
      alert(s1,e :: _,s2)

  rule string_append_list(s1) => s1' & print s1' &
      print "<unprintable>" &
      string_append_list(s2) => s2' & print s2'
      -----------------------------------------
      alert(s1,e :: _,s2)

  rule string_append_list(s1) => s1' & print s1' &
      string_append_list(s2) => s2' & print s2'
      -----------------------------------------
      alert(s1,[],s2)
end
```

B.4. C Interface to the Solver

Since RML compiles rules to C, we can interface with the solver by implementing a module entirely in C, and only declaring its rules in RML to link other RML modules against it.

```
module Solver:
 with "babelsberg.rml"
 (* Call solver in C *)
 relation solve: (string, string) => babelsberg.Program list
 relation string_real: string => real
end
```

```
/* Glue to call parser (and thus scanner) from RML */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include "rml.h"
#include "babelsberg.h"
#include "parser.h"

extern int yy_scan_string(char *yy_str);
extern void* absyntree;

/* No init for this module */
void Solver_5finit(void) {}
/* The glue function */
RML_BEGIN_LABEL(Solver__solve)
{
   char *assignment;
   double rvalue = 0;
   char *first_param = RML_STRINGDATA(rmlA0);
   char *second_param = RML_STRINGDATA(rmlA1);
   char c;

   printf("\n\n### These are the current constraints: %s\n", first_param);

   FILE* z3 = fopen("constraints.smt", "w");
   fwrite(second_param, sizeof(char), strlen(second_param), z3);

   /* Call the alert relation, so we only print this in debug mode */
   rml_state_ARGS[0]= mk_cons(mk_scon(
        "\n\nA terminal with your $BBBEDITOR will open. Please enter a new " \
        "environment satisfying the constraints as 'var := value' pairs, "\
        "each separated by a newline. Save and close finishes.\n"\
        "To fail in the solver, just write 'unsat'.\n\n"), mk_nil());
   rml_state_ARGS[1]= mk_nil();
   rml_state_ARGS[2]= mk_nil();
   rml_prim_once(RML_LABPTR(babelsberg__alert));
   fflush(NULL);

   int exitcode = system("$BBBEDITOR input");
```

```
  if (exitcode != 0) {
     RML_TAILCALLK(rmlFC);
  }

  FILE* input = fopen("input", "r");
  void* list = mk_nil();

  while(fscanf(input, "%m[()#-{}a-zA-Z0-9.:=, \"]\n", &assignment) != EOF) {
     /* printf("%s\n", assignment); */
     yy_scan_string(assignment);
     if (yyparse() != 0) {
        fprintf(stderr, "Parsing model failed!\n");
        RML_TAILCALLK(rmlFC);
     }
     list = mk_cons(absyntree, list);

     free(assignment);
  }

  rmlA0 = list;
  RML_TAILCALLK(rmlSC);
}
RML_END_LABEL

RML_BEGIN_LABEL(Solver__string_5freal)
{
  char *first_param = RML_STRINGDATA(rmlA0);
  char *endptr = first_param;
  errno = 0;
  double r = strtod(first_param, &endptr);

  if (endptr == first_param || errno != 0) {
     /* printf("Conversion failed %s\n", first_param); */
     RML_TAILCALLK(rmlFC);
  }
  /* printf("Conversion success %s —> %f\n", first_param, r); */
  rmlA0 = mk_rcon(r);
  RML_TAILCALLK(rmlSC);
}
RML_END_LABEL
```

To run the test with automatically calling the Z3 binary and feeding the solution back to the RML executable, we use the following additional Ruby script. When the environment variable BBBEDITOR is set to this script, the C interface that implements the solver judgment will call this file with the generated constraints.

```
module Z3ModelParser
  extend self

  def parse(str)
    raise "no model!" unless str.start_with?("(model")
    env = read_environment(str)
    simple_env = simplify_env(env)
    remove_temps(simple_env)
  end

  def hash_to_rml_env(hash)
    hash.each_pair.inject([]) do |acc,kv|
      k = kv[0]
      if k == "H"
        acc + [kv[1].inject([]) do |heapacc, object|
          ref = object[0]
          key = value_to_rml(object[1])
        heapacc + ["H(#{ref}) := #{key}"]
      end.join("\n")]
    elsif k =~ /^(?:[vi]Rec|invalidR)$/
      acc
    else
      v = value_to_rml(kv[1])
      acc + ["#{k} := #{v}"]
    end
  end.sort_by {|a| a =~ /(\d+)/; $1.to_i }.reject(&:empty?).join("\n")
end

def value_to_rml(v)
  if Hash === v
    "{" + v.inject([]) do |acc2,kv2|
      acc2 + ["#{kv2[0]}: #{value_to_rml(kv2[1])}"]
    end.sort.join(", ") + "}"
  else
    v
  end
end

def remove_temps(env)
  Hash[*env.each_pair.map do |key, value|
    if key =~ /!/
      nil
    else
      [key, value]
    end
  end.flatten.compact]
end

def simplify_env(env)
  Hash[*env.each_pair.map do |key, value|
    [key, simplify_value(env, value)]
  end.flatten.compact]
end

def simplify_value(env, value)
  if value =~ /^\s*\(Real (\d+\.\d+)\)\s*$/
    $1
  elsif value =~ /^\s*\(Real \(\- (\d+\.\d+)\)\)\s*$/
    "-#{$1}"
  elsif value =~ /^\s*\(\- (\d+\.\d+)\)\s*$/
    "-#{$1}"
  elsif value =~ /^\s*\(Reference ref(\d+)\)\s*$/
    "##{$1}"
  elsif value =~ /^\s*\(Reference nil\)\s*$/
    "nil"
  elsif value =~ /^\s*\(Bool (\w+)\)\s*$/
    $1
  elsif value =~ /^\s*\(Record \(_ as-array (k!\d+)\)\)\s*$/
    transform_ite(env, env[$1])
  elsif value =~ /^\s*\(_ as-array (k!\d+)\)\s*$/
    transform_ite(env, env[$1])
  elsif value =~ /ite/
    transform_ite(env, value)
  elsif value =~ /^\s*\(Record \(store/
    transform_store(env, value)
  else
    value
  end
end

def transform_store(env, str)
```

```ruby
    record = {}
    regexp = /([\w]+) (\d+\.\d+)/
    s = str
    while md = regexp.match(s)
      s = md.post_match
      next if md[1] == "undef"
      record[simplify_value(env, md[1])] = simplify_value(env, md[2])
    end
    record
  end

  def transform_ite(env, str)
    record = {}
    regexp = /ite \(= \w!\d (\(?[\w\s\d]+\)?)\) ([^\n]+)\n/m
    s = str
    while md = regexp.match(s)
      s = md.post_match
      next if md[1] == "undef"
      record[simplify_value(env, md[1])] = simplify_value(env, md[2])
    end
    record
  end

  def read_environment(str)
    env = {}
    pos = "(model".size
    while pos < str.size
      pos, fun = read_fun(pos, str)
      key, value = parse_fun(fun)
      env[key] = value
    end
    env
  end

  def read_fun(pos, str)
    i = str.index("(", pos)
    return str.size, nil if i.nil?
    open = 1
    ((i + 1)..str.size).each do |idx|
      c = str[idx]
      case c
      when "(" then open += 1
      when ")" then open -= 1
      end
      return idx, str[i..idx] if open == 0
    end
    return str.size, nil
  end

  def parse_fun(sexp)
    sexp =~ /^\(define-fun ([^ ]+) (?:\((?:\([^\)]+\))?\)) (?:[A-Za-z]+|\([^\)]+\))?\s+(.*)\)$/m
    return $1, $2
  end

end

outfile = ARGV[0]


tree = Hash[*ps -eo pid,ppid.scan(/\d+/).map{|x|x.to_i}]
shid = Process.ppid
bbbid = tree[shid]
catid = tree[bbbid]
```

```ruby
rakeid = tree[catid]
commandline = %x"ps -o cmd -fp #{catid}".lines.to_a.last
example = /cat\s+(.*\d+[a-z]?\.txt)\s+/.match(commandline)[1]
solution = example.sub(/\.txt$/, ".env")
number = solution.split("/").last.sub(/\..*$/,"")

begin
  environments = File.read(solution).split(";\n")
rescue Exception
  environments = []
end


if File.exist?(outfile) && File.read(outfile) =~ /^cIIndex := (\d+)$/m
  idx = $1.to_i
else
  idx = 0
end

File.open(outfile, 'w') do |f|
  f << "cIIndex := #{idx + 1}\n"
  if ENV["BBBReview"] || ENV["BBBZ3"]
    puts "\n# #{number} # This is the expected solution:\n#{environments[idx]}"
    if ENV["BBBZ3"]
      puts "\n# #{number} # This is what Z3 produces:"
      system("#{File.expand_path('../../z3', outfile)} -smt2 constraints.smt > constraints.
           model")
      output = File.read("constraints.model")
      if midx = output.index("(model")
        hash = Z3ModelParser.parse(output[midx..-1])
        model = Z3ModelParser.hash_to_rml_env(hash)
        puts model
        ENV["BBBZ3FB"] ? f << environments[idx] : f << model + "\n"
      else
        puts output
      end
    else
      f << environments[idx]
    end
    unless ENV["BBBZ3Auto"]
      %x{xterm -e "read -p 'Please review the constraints and solution. Are they ok? (Y/n)' -
           n 1 -r; echo; if [[ \\$REPLY =~ ^[Nn]$ ]]; then kill #{rakeid}; fi"}
    end
    if ENV["BBBZ3AutoCompare"]
      shortmodel = model.lines.map { |l| l.gsub(" ", "").strip }
      shortmodel += shortmodel.map do |l|
        l.gsub(".0", "") # use ints
      end
      environments[idx].lines.each do |line|
        next if line.strip.empty?
        unless shortmodel.include? line.gsub(" ", "").strip
          e = "ERROR: Z3 model does not check out! #{line.gsub(" ", "")} not found in #{
               shortmodel}"
          puts e
          f << e
        end
      end
    end
  else
    f << environments[idx]
  end
end
```

C. Test Generator

The language independent part of the test suite generation comprises another set of scripts. `find_example.rb` is used to determine which example program is currently running, `assert.rb` is used to generates assertions from the expected results of the examples.

```
#!/usr/bin/env ruby

module FindExample
  extend self

  def example
    return @example if @example
    tree = Hash[*ps -eo pid,ppid.scan(/\d+/).map{|x|x.to_i}]
    shid = Process.ppid
    bbbid = tree[shid]
    catid = tree[bbbid]
    rakeid = tree[catid]
    commandline = %x"ps -o cmd -fp #{catid}".lines.to_a.last
    @example = /cat\s+(.*\d+[a-z]?\.txt)\s+/.match(commandline)[1]
  end

  def solution
    example.sub(/\.txt$/, ".env")
  end

  def id
    example.sub(/(?:.*?)(\d+[a-z]*)\.txt$/, "\\1")
  end
end

print FindExample.send(ARGV[0]) if __FILE__ == $0 && ARGV[0]
```

```
#!/usr/bin/env ruby

require File.expand_path("../find_example", __FILE__)

outfile = ARGV[0]
index = ARGV[1].to_i
solution = FindExample.solution

begin
  environments = File.read(solution).split(";\n")
rescue Exception
  environments = []
end

File.open(outfile, 'w') do |f|
  if env = environments[index]
    identities = {}
    objects = {}
    # remove trailing numbers
    env.gsub!(/([^ 0-9]+)\d+ :=/, "\\1 :=")
    env = env.lines.map do |line|
      if line =~ /^H\(#(\d+)\) := (.*)$/
        objects[$1] = $2
        nil
      else
        line
      end
    end.compact.map do |line|
      if line =~ /^([^ ]+) := #(\d+)$/
        if identities[$2]
          "#{$1} := H(#{identities[$2]})\n"
        else
          identities[$2] = $1
          "#{$1} := #{objects[$2]}\n"
        end
      else
        line
      end
    end.join

    env.gsub!(/^unsat$/, "unsat := true")

    f << env
  else
    f << "unsat := true"
  end
end
```

These scripts are called from the `Assertions` RML module.

```
module Assertions:
  relation assert: int => ()
  relation exampleId: () => string
end
```

```
/* Glue to call parser (and thus scanner) from RML */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include "rml.h"
#include "javascript.h"
#include "../objects/parser.h"


extern int yy_scan_string(char *yy_str);
extern void* absyntree;

int ciindex = -1;

/* No init for this module */
void Assertions_5finit(void) {}
/* The glue function */
RML_BEGIN_LABEL(Assertions__assert)
{
    int first_param = RML_UNTAGFIXNUM(rmlA0);
    char *assignment;
    char cmd[255];
    ciindex += first_param;
    snprintf(cmd, 255, "$BBBPATH/assert.rb input %d", ciindex);
    int exitcode = system(cmd);
    if (exitcode != 0) {
        RML_TAILCALLK(rmlFC);
    }

    FILE* input = fopen("input", "r");
    while(fscanf(input, "%m[()#-{}a-zA-Z0-9.:=, \"]\n", &assignment) != EOF) {
        /* printf("%s\n", assignment); */
        yy_scan_string(assignment);
        if (yyparse() != 0) {
            fprintf(stderr, "Parsing model failed!\n");
            RML_TAILCALLK(rmlFC);
        }
        rml_state_ARGS[0] = absyntree;
        rml_prim_once(RML_LABPTR(javascript__printassert));
        free(assignment);
    }
    rmlA0 = mk_nil();
    RML_TAILCALLK(rmlSC);
}
RML_END_LABEL

RML_BEGIN_LABEL(Assertions__exampleId)
{
    FILE* stream = popen("$BBBPATH/find_example.rb id", "r");
    char* id[5] = {'\0'};
    int read = fread((void*)id, 4, sizeof(char), stream);
    pclose(stream);

    rmlA0 = mk_scon((const char*)id);
    RML_TAILCALLK(rmlSC);
}
RML_END_LABEL
```

To generate the test suites requires rules for the translation and a scaffolding file, which have to be written by the developer. Below, we give the scaffolding for the JavaScript language. Note that this includes the `babelsberg.rml` file, and just adds the translation mechanism.

```
module javascript:
  with "../objects/babelsberg.rml"
  with "assertions.rml"

  relation printprogram: babelsberg.Program => ()
  relation printassert: babelsberg.Program => ()
end

relation printprogram: babelsberg.Program => () =
  rule printPreamble() &
        printS(statement, " ") &
        printPostscript()
        -----------------------------------------
        printprogram(babelsberg.PROGRAM(statement))
end

relation printassert: babelsberg.Program => () =
  rule print " this.assert(" &
        printE(babelsberg.LVALUE(l)) & print " === " & printE(e) &
        print ");\n"
        ---------------------------------------------------------------------
        printassert(babelsberg.PROGRAM(babelsberg.ASSIGN(l,babelsberg.DEREF(e))))

  rule print " this.assert(this.fieldEquals(" &
        printE(babelsberg.LVALUE(l)) & print ", " & printE(babelsberg.IRECORD(fieldexps)) &
        print "));\n"
        ------------------------------------------------------------------------------
        printassert(babelsberg.PROGRAM(babelsberg.ASSIGN(l,babelsberg.IRECORD(fieldexps))))

  rule print " this.assert(" &
        printE(babelsberg.LVALUE(l)) & print " == " & printE(e) &
        print ");\n"
        -------------------------------------------------------
        printassert(babelsberg.PROGRAM(babelsberg.ASSIGN(l,e)))
end

```

```
relation printPreamble: () => () =
 rule print "test" &
      Assertions.exampleId() => s & print s &
      print ": function() {\n" &
      print " delete bbb.defaultSolver;\n" &
      print " bbb.defaultSolvers = [new CommandLineZ3(), new DBPlanner()];\n" &
      print " bbb.defaultSolvers[1].$$identity = true;\n" &
      print " var ctx = {unsat: false};\n\n"
      --------------------------------------------------------------------------
      printPreamble()
end

relation printPostscript: () => () =
 rule print "},\n"
      -----------------
      printPostscript()
end

relation printS: (babelsberg.Statement, string) => () =
 axiom printS(babelsberg.SKIP, i) => ()

 rule printS(s1, i) & printS(s2, i)
      -------------------------------
      printS(babelsberg.SEQ(s1,s2), i)

 rule string_append(i, " ") => i' &
      print i & print "if (" & printE(e) & print ") {\n" &
      printS(s1, i') &
      print i & print "} else {\n" &
      printS(s2, i') &
      print i & print "}\n"
      ----------------------------------
      printS(babelsberg.IF(e, s1, s2), i)

 rule string_append(i, " ") => i' &
      print i & print "while (" & printE(e) & print ") {\n" &
      printS(s, i') &
      print i & print "}\n"
      --------------------------------
      printS(babelsberg.WHILE(e, s), i)

 rule print i & print "try {\n" &
      print i & print " " &
         printE(babelsberg.LVALUE(l)) & print " = " & printE(e) & print ";\n" &
      print i & print "} catch (e) { ctx.unsat = true }\n" &
      Assertions.assert(2)
      ------------------------------------------------------------------------
      printS(babelsberg.ASSIGN(l,e), i)

 rule print i & print "try { once: {\n" &
      print i & print " " & printE(babelsberg.IDENTITY(e0, e1)) & print "\n" &
      print i & print "} } catch (e) { ctx.unsat = true }\n"
      ------------------------------------------------------------------------------
      printS(babelsberg.ONCE(
         babelsberg.CONSTRAINT(babelsberg.REQUIRED, babelsberg.IDENTITY(e0, e1))), i)

 rule print i & print "try { always: {\n" &
      print i & print " " & printE(babelsberg.IDENTITY(e0, e1)) & print "\n" &
      print i & print "} } catch (e) { ctx.unsat = true }\n"
      ------------------------------------------------------------------------------
      printS(babelsberg.ALWAYS(
         babelsberg.CONSTRAINT(babelsberg.REQUIRED, babelsberg.IDENTITY(e0, e1))), i)

 rule print i & print "try { once: {\n" &
```

```
      print i & print " " & print "priority: '" & printRho(rho) & print "';\n" &
      print i & print " " & printE(e) & print "\n" &
      print i & print "} } catch (e) { ctx.unsat = true }\n" &
      Assertions.assert(1)
      ----------------------------------------------------------------------------
      printS(babelsberg.ONCE(babelsberg.CONSTRAINT(rho,e)), i)

  rule print i & print "try { always: {\n" &
      print i & print " " & print "priority: '" & printRho(rho) & print "';\n" &
      print i & print " " & printE(e) & print "\n" &
      print i & print "} } catch (e) { ctx.unsat = true }\n" &
      Assertions.assert(1)
      ----------------------------------------------------------------------------
      printS(babelsberg.ALWAYS(babelsberg.CONSTRAINT(rho,e)), i)
end

relation printE: babelsberg.Exp => () =
  rule print "true"
      ------------------------------------------------------
      printE(babelsberg.VALUE(babelsberg.K(babelsberg.TRUE)))

  rule print "false"
      -------------------------------------------------------
      printE(babelsberg.VALUE(babelsberg.K(babelsberg.FALSE)))

  rule print "null"
      -----------------------------------------------------
      printE(babelsberg.VALUE(babelsberg.K(babelsberg.NIL)))

  rule print s
      -----------------------------------------------------------
      printE(babelsberg.VALUE(babelsberg.K(babelsberg.STRING(s))))

  rule real_string(r) => s &
      print s
      ---------------------------------------------------------
      printE(babelsberg.VALUE(babelsberg.K(babelsberg.REAL(r))))

  rule string_length(x) => i &
      i - 1 => li &
      string_nth(x, li) => lst &
      lst = #"?" &
      string_list(x) => chars &
      list_delete(chars, li) => chars' &
      list_string(chars') => xs &
      print "ro(ctx." & print xs & print ")"
      ------------------------------------------------
      printE(babelsberg.LVALUE(babelsberg.VARIABLE(x)))

  rule print "ctx." & print x
      ------------------------------------------------
      printE(babelsberg.LVALUE(babelsberg.VARIABLE(x)))

  rule print "(" & printE(e) & print ")" &
      print "." & print l
      ------------------------------------------------
      printE(babelsberg.LVALUE(babelsberg.FIELD(e, l)))

  rule printE(e1) & printOp(op) & printE(e2)
      ------------------------------------
      printE(babelsberg.OP(e1, op, e2))

  rule printE(e1) & print " === " & printE(e2)
      --------------------------------------
      printE(babelsberg.IDENTITY(e1, e2))

  rule l = "Testpointxequals5" & (* special treatment, value class *)
      Assertions.exampleId => id & id = "47" &
      print "this.Testipointxequals5" &
      print "(" & printEachArg(e :: argexps) & print ")"
      ----------------------------------------------------------
      printE(babelsberg.CALL(e,l,argexps))

  rule print "this." & print l &
      print "(" & printEachArg(e :: argexps) & print ")"
      ------------------------------------------------
      printE(babelsberg.CALL(e,l,argexps))

  rule print "new Object({" & printFields(fieldexps) & print "})"
      ------------------------------------------------------
      printE(babelsberg.UIDRECORD(fieldexps))

  rule print "Object({" & printFields(fieldexps) & print "})"
      --------------------------------------------------
      printE(babelsberg.IRECORD(fieldexps))
end

relation printEachArg: babelsberg.Exp list => () =
  axiom printEachArg([]) => ()

  rule printE(e)
      ---------------------
      printEachArg(e :: [])

  rule printE(e) & print ", " &
      printEachArg(rest)
      -----------------------
      printEachArg(e :: rest)
end

relation printFields: babelsberg.ObjectLiteral => () =
  axiom printFields([]) => ()

  rule print l & print ": " & printE(e)
      --------------------------------
      printFields((l,e) :: [])

  rule print l & print ": " & printE(e) &
      print ", " & printFields(rest)
      -----------------------------------
      printFields((l,e) :: rest)
end

relation printOp: babelsberg.Op => () =
  rule print " + "
      -----------------------
      printOp(babelsberg.ADD)

  rule print " - "
      -----------------------
      printOp(babelsberg.SUB)

  rule print " / "
      -----------------------
      printOp(babelsberg.DIV)

  rule print " * "
      -----------------------
      printOp(babelsberg.MUL)

  rule print " < "
      ----------------------------
      printOp(babelsberg.LESSTHAN)

  rule print " <= "
      --------------------------
      printOp(babelsberg.LEQUAL)

  rule print " == "
      -------------------------
      printOp(babelsberg.EQUAL)

  rule print " != "
      --------------------------
      printOp(babelsberg.NEQUAL)

  rule print " >= "
      --------------------------
      printOp(babelsberg.GEQUAL)

  rule print " > "
      -------------------------------
      printOp(babelsberg.GREATERTHAN)

  rule print " && "
      -----------------------
      printOp(babelsberg.AND)

  rule print " || "
      ----------------------
      printOp(babelsberg.OR)
end

relation printRho: babelsberg.Rho => () =
  rule print "weak"
      -------------------------
      printRho(babelsberg.WEAK)

  rule print "medium"
      ---------------------------
      printRho(babelsberg.MEDIUM)

  rule print "required"
      -----------------------------
      printRho(babelsberg.REQUIRED)
end
```

ISBN 978-3-86956-348-0
ISSN 1613-5652