Journal of Information Processing Vol.24 No.6 917–927 (Nov. 2016)

[DOI: 10.2197/ipsjjip.24.917]

Regular Paper

Declaring Constraints on Object-oriented Collections

Tim Felgentreff[1,a)] Robert Hirschfeld[1,b)] Maria Graber[1,c)]
Alan Borning[2,d)] Hidehiko Masuhara[3,e)]

Received: January 28, 2016, Accepted: April 25, 2016

Abstract: Logic puzzles such as Sudoku are described by a set of properties that a valid solution must have. Constraints are a useful technique to describe and solve for such properties. However, constraints are less suited to express imperative interactions in a user interface for logic puzzles, a domain that is more readily expressed in the object-oriented paradigm. Object constraint programming provides a design to integrate constraints with dynamic, object-oriented programming languages. It allows developers to encode multi-way constraints over objects using existing, object-oriented abstractions. These constraints are automatically maintained at run-time. In this paper we present an application of this design to logic puzzles in the Squeak/Smalltalk programming environment, as well as an extension of the design and the formal semantics of Babelsberg to allow declaring constraints using the imperative collection API provided in Squeak. We argue that our implementation facilitates creating applications that use imperative construction of user interfaces and mutable program state as well as constraint satisfaction techniques for different parts of the system. The main advantage of our approach is that it moves the burden to maintain constraints from the developer to the runtime environment, while keeping the development experience close to the purely object-oriented approach.

Keywords: object constraint programming, constraint imperative programming, constraint solving, babelsberg

1. Introduction

Logic puzzles are declarative. Their rules declare *what* a valid solution should look like, and they can then be solved without any pre-described algorithm other than logical deduction techniques. A famous example is Sudoku. The rules of a logic puzzle describe properties that should be maintained while solving the puzzle. For example, in Sudoku, the properties are that each row, column, and box contain the numbers from 1 to 9 exactly once. The properties of a logic puzzle can be formulated as formal constraints, which a constraint solver can use to find one or more solutions or to check if a solution input by the user is valid [12].

Babelsberg [5] is a design to integrate constraints into object-oriented languages in a way that allows programmers to dynamically create and satisfy constraints on objects. The design is a strict extension of the object-oriented semantics of the underlying host language. Babelsberg uses object-oriented method definitions to define constraints rather than a constraint (DSL) [17], [19]. As a consequence, Babelsberg respects encapsulation and object-oriented abstractions. The design also supports solver features such as constraint priorities [2] and incremental resolving [8]. Recently, the design has been extended to allow multiple constraint solvers to cooperate to find a solution [6].

This design lends itself well to build interactive user interfaces for logic puzzles where the puzzle rules are expressed as constraints on the Morphic objects. User interface frameworks such as *Morphic* [15] are inherently imperative – the user interface consists of compositions of *Morphs* that have state and react to user input events. Morphic was first implemented in Self, with later implementations in Squeak [16] and JavaScript [20].

In a standard imperative programming language, constraint solving and satisfaction is implemented explicitly. Using just Morphic in a standard imperative language, developers have to ensure that all event sources that might change the user interface trigger calls to resatisfy constraints in some way. In contrast, Babelsberg maintains constraints automatically, regardless of how the system was perturbed. This reduces the amount of knowledge the developer has to have about possible event sources for the Morphs. We argue that this is more in line with the encapsulation and abstraction desired in object-oriented applications.

An incomplete aspect of the original Babelsberg design was that it only allowed constraints on objects and their parts, but did not allow multi-directional solving for constraints on collections. In the context of logic puzzles the rules are usually defined on sets of objects (for example, Sudoku constraints are defined on rows, columns, and boxes.) In prior work, we experimented with an extended Squeak/Smalltalk based prototype implementation of the Babelsberg design — Babelsberg/S — to support operations on collections of objects [9]. In this paper, we present a general design from this prototype implementation, as well as semantic rules to supplement the existing Babelsberg semantics [7].

Thus, the contributions of this work are:

- We describe an implementation of the Babelsberg design in

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
[2] University of Washington, Seattle, WA, USA
[3] Department of Mathematical and Computing Science, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
a) tim.felgentreff@hpi.uni-potsdam.de
b) robert.hirschfeld@hpi.uni-potsdam.de
c) maria.graber@student.hpi.uni-potsdam.de
d) borning@cs.washington.edu
e) masuhara@acm.org

Squeak/Smalltalk.
- We describe an extension to Babelsberg that let the programmer conveniently specify constraints on collections that works even if the underlying solver does not support collection types.
- We present a technique for Morphic applications to interact with constraints, using as a running example an interactive Sudoku application where constraints are resolved in response to user input, and constraint solving affects the user interface display.
- In an appendix, we present semantic rules to supplement the formal Babelsberg design to support collection predicates. To simplify the rules for the semantics, however, we assume the solver supports uninterpreted functions to represent field access (but not collections).

2. Object Constraint Programming in Squeak

This section describes how constraints are expressed in our Squeak implementation of Babelsberg, called Babelsberg/S. For our examples, we use the rules of a Sudoku puzzle.

```
constraint := [
  (sudoku at: 1 at: 1) between: 1 and: 9
] alwaysSolveWith: solver.
```

Listing 1: Defining the domain of a Sudoku cell.

Listing 1 shows the constraint for defining the domain of one Sudoku cell. In general, a constraint in Babelsberg/S is specified as a block that evaluates to a boolean — if the block evaluates to `true`, the constraint is satisfied. As mentioned in Section 1, this block contains Smalltalk code, rather than code written in a separate DSL. The variable `sudoku` in Listing 1 represents the grid of cells in the interactive application from the outer scope and the method `between:and:` is a predefined predicate on Squeak numbers that just checks whether the receiver's value is between the upper- and lower-bound arguments. To actually turn this code in into a constraint that can be handed to a solver, we send the message `alwaysSolveWith:` to the block, passing as argument an instance that implements the interface of the Bablelsberg/S `ConstraintSolver` class. (It is also possible to solve the constraint with a default constraint solver, which is global inside the Squeak image, by sending `alwaysTrue`.) While the Smalltalk block can contain arbitrary Smalltalk code, asking the system to interpret it as a constraint puts the same restrictions on the expressions insides the block as for previous implementations of Babelsberg [5], [6]. These are a) an expression that is used as a constraint must evaluate to a boolean (the constraint is that it evaluate to true), b) the expression should return the same result on repeated evaluation (so that, for example, a random number generator would not qualify), and c) the expression should be free of side-effects.

Translating Constraints in Babelsberg/S

To translate the Smalltalk expression into a form suitable for a constraint solver, the constraint block is executed in a different execution mode called *constraint construction mode* which uses symbolic execution [3], [13] to create constraint expressions from

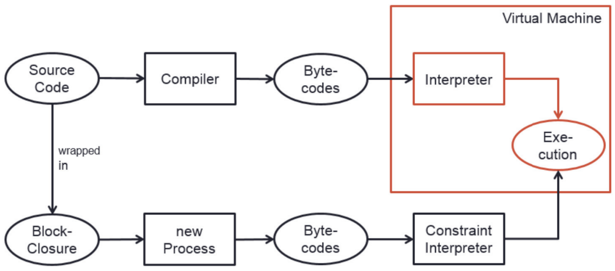


Fig. 1 The architecture of Babelsberg/S constraint construction mode.

the code. The block is only evaluated in constraint construction mode when either `alwaysTrue` or `alwaysSolveWith:` are sent to it, otherwise it is just an ordinary Squeak block.

Squeak/Smalltalk includes an in-image Smalltalk interpreter that we instrumented to implement our constraint construction mode. The resulting architecture is shown in **Fig. 1**. Squeak stack frames can be reified into instances of subclasses of the `ContextPart` class. These provide methods to interpret each bytecode, a facility used by the Squeak debugger. Babelsberg/S uses the instruments this interpretation to evaluate the bytecodes in the constraint block. The `alwaysTrue` method creates a new Process (a Smalltalk green-thread) that is interpreted stepwise using the interface of the `ContextPart` objects. Where interpretation in constraint construction mode deviates from normal Smalltalk semantics, we use ContextS [10] to instrument methods whose behavior needs to change inside a constraint construction mode layer.

Consider the constraint in Listing 1: the block `[(sudoku at: 1 at: 1) between: 1 and: 9]` is compiled into bytecode. A new Squeak process is created (but not scheduled) by sending the method `newProcess` to it. The process has a stack with exactly one frame (a `ContextPart` object.) That frame's program counter is set to 0 and it contains the bytecode for the constraint block. The Babelsberg/S interpreter then steps through this frame by interpreting the bytecodes one by one, including doing method lookup and creating new frames as needed. An important consequence of this is that a variable binding that is used as receiver in a constraint block cannot be allowed to change, because then the lookup, and thus the constructed constraint, might be invalid. Thus, for Listing 1, the solver cannot simply find a collection that already satisfies the constraint and change the binding of the `sudoku` variable. Instead, it has to change the contents of the underlying collection to satisfy the constraint. This restriction does not apply to bindings that were created during constraint construction, such as return values of methods – so the solver can (and will) change what the method `at:at:` returns when sent to `sudoku`.

The modified interpreter creates `ConstraintVariable` objects for instance variables that are accessed through accessor methods. All methods are then called on these `ConstraintVariable` objects. Operator methods such as +, -, or <= construct constraint expressions instead of evaluating directly. Other methods that the solver does not directly support are partially evaluated to break them down into these primitive operations. In the case of `between:and:`, for example, the constraint constructed from partially evaluating the method would be equivalent to specifying `n >= lower and: [n <= upper]` directly. If the method evolves, the constructed constraints automatically change as well, since they are constructed from the same underlying code. By re-using existing methods in this way, Babelsberg/S supports the object-oriented abstractions that already exist in the system. This is equivalent to the Babelsberg implementations in Ruby and JavaScript [5].

Additionally, the interpreter creates instance-specific method wrappers to intercept access to these variables. The wrappers delegate read and write access to the corresponding `ConstraintVariable`, which calls the solver as needed to keep the constraints satisfied and returns the value of the variable from the solver's solution.

In contrast to JavaScript or Ruby, Squeak/Smalltalk does not allow instance-specific behavior directly. All methods and instance variables are declared on the class. However, wrapping accessors on the class of any encountered object would cause all instances of that class in the system to go through our wrapper, which imposes considerable performance overhead. To wrap only the encountered instances, we create anonymous subclasses of their class, and use Smalltalks `become:` facility to change the class of the object to the anonymous subclass. We then install our wrappers only on this instance-specific subclass.

This solution to instance-specific behavior means that there is no run-time overhead when using objects that have no constraints on them. Constrained objects are easily discovered through Smalltalk's meta-programming interface, because their class has no name and only wraps the accessors encountered in the constraint. We encountered methods in the core system that check for the class of its arguments not using the `isKindOf:` method (which works correctly for instances of subclasses), but by directly comparing the class pointer. Although one might consider this as a bug in the method, we are working on a solution to instance-specific behavior that is completely transparent to these common uses of meta-programming.

After constraint construction has interpreted the block, the generated constraint expressions are added to a `Constraint` object, which is passed to the constraint solver. We explain the solving process in more detail in Section 3.3. If solving succeeds, the method `alwaysSolveWith:` returns the newly created constraint object. This object can then be used for reflection (e.g., to inspect which variables participate in the constraint) as well as to dynamically disable and re-enable the constraint. If solving fails, an exception is raised, which must be handled by the programmer. In that case, the constraint is not added and the system remains unperturbed.

3. Constraints on Collections of Objects

The original Babelsberg design did not support constraints on collections directly; rather, it was proposed to use a specialized solver for collections [5]. The reason for this were two-fold:

First, imperative control-flow structures like unbounded loops and early multiple returns do not work with our previous design for constraint construction mode. A core problem with constraints over collections is that, in imperative code, predicates are expressed through loops over the length of the collection. This poses problems when the length of the collection is not fixed that could lead to unbounded unrolling. In the presence of early returns in the code block of the loop, not enough of the loop might be unrolled and the system might miss possible solutions. Other constraint programming systems such as Backtalk [18] or OPL [21] deal with this by providing special collection operators such as `forall` to express constraints, and by disallowing imperative control flow statements inside of these functions. For Babelsberg, however, we find this solution undesirable since we want to support ordinary object-oriented code and re-use the existing predicate methods on collections.

Second, our original design relied on the solvers supporting the basic operations (such arithmetic) that are used in constraints. For collections, that means the solvers would have to support field access and variable sized collections, if these are available in the language. Many solvers that are used in interactive systems do not support this, however. In Babelsberg, we want to support and use many different solvers in a cooperating fashion, and we cannot rely on support for collections. A key issue in this extension to our design is thus the translation of collection predicates into constraints in a way that does not require the underlying solver to support collections.

To model an entire Sudoku puzzle, we need to assert the constraint given in Listing 1 for each cell. With the existing Babelsberg design, this would either require a solver for collections that supports domains for numbers, or alternatively, loop over the cells imperatively, as shown in Listing 2.

```
(1 to: sudoku size) do: [:index |
    [(sudoku at: index) between: 1 and: 9]
        alwaysSolveWith: solver].
```

Listing 2: Defining the domain of Sudoku cells with a loop.

The code has two main problems, however. First, if we consider collections that can grow (or shrink), these constraints would then be incorrect — they would either have to be redacted and the loops re-executed or after adding the above constraints once any changes to the size of the collection must be prohibited. Second, many object-oriented languages including Squeak/Smalltalk come with APIs to work with collections, and rather than iterating manually, a method such as `allDifferent`, if available on the `Collection` class, should work in a constraint, because it satisfies our restrictions on constraint expressions that they return a boolean and are free of side-effects. The developer should not have to know that this method encapsulates a loop over the collection to decide if it can be used in constraints:

```
[collection allDifferent] alwaysTrue.
```

The formal design of Babelsberg indeed does support such methods, but only for solving in the *forward direction*, that is, to execute them and use the result as a constant [7]. In this case, solving in the forward direction would be of little use, however, since if the result of the call to `allDifferent` is not already true, there is nothing the system can do, since the result when treated as a constant is simply false.

Supporting collections in constraints more directly is thus useful in this application, and more generally in any application that deals with finite domain problems as well as representations of those problems (graphical or otherwise) that are more readily expressed using imperative code. This combination makes a Sudoku application a good example of the kinds of applications we want to support with our design.

There are a number of collection predicates that are commonly used in constraints and that would be useful to support. For these, we propose that implementers of Babelsberg-like languages must check their actual implementation and decide for each if they should allow them in a modified form of constraint construction mode. In this mode, rather than simply executing through complex methods involving loops, we convert any operations involving collection elements that are used as tests into constraints. Any indexing variable is treated as read-only. If the test would trigger an early return, we ignore the return and continue.

Depending on the type of recognized test, the generated constraints must then be added to a conjunction or disjunction. The system determines whether to use a conjunction or disjunction if the method uses an early return optimization. If, depending on an element test, the method would early return `true`, the tests must be combined in a disjunction, since it is enough to satisfy just one test to have the method return the same. Otherwise, the elements are negated and added in to a conjunction. Thus, an implementation of `allDifferent` as in Listing 3 would be turned into a conjunction of pair-wise inequality constraints.

```
allDifferent
  1 to: self length do: [:i |
    1 to: self length do: [:j |
      (i ~= j and: [(self at: i) = (self at: j)])
        ifTrue: [↑ false]]]
  ↑ true
```

Listing 3: A possible implementation of `allDifferent`.

The code for `allDifferent` would be expanded into a conjunction of constraints, because the early return is `false`. The constraints in the conjunction would be for the tests to that early return, pair-wise constraining `(self at: i) = (self at: j)` to be false (with the values of `i` and `j` fixed for each constraint). In addition, supposing the `length` method represents a field access, this field is also used in the constraint, and thus the system can track any change to this field to trigger regenerating the constraints on the collection.

Some common predicates available on collections in Squeak/Smalltalk are translated as per **Table 1**. Note that both the test if some element satisfies a particular predicate as well as the test for membership—the latter being a special case of the former—require a disjunction. Without any other constraints, and since our design does not use Prolog-style backtracking, they would probably always be satisfied by setting the first—or last, depending on the concrete implementation—element of the array. Even though this list contains only a few predicates, in practice many languages come only with a small set of primitive collection types that are supported at a language level. Languages with a rich collection library such as Common Lisp or Squeak/Smalltalk [11] are built around a small number of types and primitive operations to access and store indexed elements in an object. Thus, implementing the special support needed to provide these basic predicates enables their use in a variety of contexts, including methods that are built on top of these predicates.

Table 1 Mapping collection predicates to declarative representation.

`anySatisfy:`	$\exists x \in \text{array} : f(x)$
`noneSatisfy:`	$\forall x \in \text{array} : \neg f(x)$
`allSatisfy:`	$\forall x \in \text{array} : f(x)$
`includes:`	$y.\exists x \in \texttt{array}.x = y$

An advantage of this general approach to supporting constraints on collections multi-directionally is that we can now also express parts of the Sudoku user interface using constraints using only the existing methods available for Morphs. For example, to ensure that the cells always have the correct size to fill the Sudoku UI, we could use the constraint given in Listing 4. Using this constraint, we abstract from the division operation on 2d points (the return value of the `extent` method for Morphs) and we allow the user to manipulate both the size of the Sudoku UI as well as the size of a single cell, and have the rest of the user interface react to update the UI consistently.

```
[sudokuUi submorphs allSatisfy: [:m |
    m extent = (sudokuUi extent / 9)]] alwaysTrue.
```

Listing 4: Constraining graphical cell sizes.

3.1 Constraints on User-Defined Methods

We do not intend for language implementers to support every possible method that a collection may have in a practical implementation, in particular if that collection may be extended with user-defined methods. As an example, consider an iterative `sum` method as in Listing 5.

```
sum
  | answer |
  answer := 0.
  1 to: self length do: [:i |
    answer := answer + (self at: i)].
  ↑ answer
```

Listing 5: A possible implementation of `sum`.

We can of course use this method in the forward direction in a constraint:

```
a := Array new: 2.
a at: 1 put: 10.
a at: 2 put: 20.
s := 30.
[s = a sum] alwaysTrue.
a at: 1 put: 100.
```

After the constraint is executed, s is still 30 (since the constraint is already satisfied); then after setting the first element of a to 100, s becomes 120. However, the method doesn't work backwards — for example, we can't constrain the sum of the array and expect the system to update one or more elements to satisfy the constraint. So the constraint in the last line below will be too hard for the system to solve:

```
a := Array new: 2.
a at: 1 put: 10.
a at: 2 put: 20.
[50 = a sum] alwaysTrue.
```

We have found a design pattern for user code that works well in these situations that *can* provide something that works both forward and backward. Rather than using a method that returns the calculated sum of the elements, we eagerly update the sum as the array changes in a variable. This can be done by writing an ordinary method that sets up a recursive network of addition constraints over the array elements. The sum becomes an instance variable of our collection, and the implementation of that collection must take care to correctly initialize the constraint network when it is created or an element is added or removed. An example of such an initialization is given in Listing 6. The advantage for code using the collection's sum in further constraints is that it is used simply as a variable — constraints on it can work both ways, and it can even be assigned and the array changes to satisfy the constraints.

```
initializeSum
  self.length = 0
    ifTrue: [[self sum = 0] alwaysTrue]
    ifFalse: [[self sum = (self at: 1) +
                   self allButLast sum] alwaysTrue].
```

Listing 6: A possible initialization for a constrainable sum.

3.2 Implementing Constraints on Collections in Babelsberg/S

Babelsberg/S implements a prototype of our scheme to support constraints on object-oriented collections. As a result, the domain constraint of a Sudoku puzzle can be expressed through sending the collection predicate `allSatisfy:` to `sudoku` (Listing 7).

```
[sudoku allSatisfy: [:cell | cell between: 1 and: 9]]
    alwaysSolveWith: solver.
```

Listing 7: Defining the domain of all Sudoku cells with the Collection API.

The extension to support collections directly in Babelsberg/S leverages the fact that Smalltalk comes with only one fixed-size pointer array type, upon which the Smalltalk collections library builds. This type provides three methods implemented in primitives for all low-level access: `at:`, `at:put:`, and `replaceFrom:to:with:startingAt:`.

Babelsberg/S subclasses the basic `Array` class and overrides the three low-level access methods to intercept any modifications to the array. In addition, it overrides the `copyFrom:to:` method, which is regularly used in Squeak to access sub-sequences of an array.

In constraint construction mode, any array that is visited in the dynamic extent of the execution is transparently replaced by the Babelsberg/S subclass. Besides the overridden methods, this subclass is a completely transparent proxy. Each element in the array that participates in the constraint is wrapped in a `ConstraintVariable`. The predicates of the collection API are straightforward to support. The predicates `anySatisfy:`, `noneSatisfy:`, and `allSatisfy:` are mapped as per Table 1. Note that for the first relation, a disjunction over all elements must be created. For solvers that do not support disjunctions, Babelsberg/S forces the first element to satisfy the block. This prevents the system from finding solutions in many cases. To find additional solutions with solvers without disjunctions requires backtracking in the case of unsatisfiable constraints. This is not implemented yet, but is left for future work.

In general, any predicate method available on collections can be used in constraints, as per our design for supporting user defined methods as collection predicates. For example, predicate methods such as `allDifferent:` can be mapped to pair-wise inequalities by simply interpreting their implementation in constraint construction mode. Other methods that are useful in constraints reduce all elements of a collection and then express properties over those reductions. Reduction methods include the `sum` method mentioned above, as well as the `count:` method that returns the number of elements that satisfy a particular condition.

The constraints created with these methods are reconstructed when the elements in the array change, but since the size of arrays is fixed, the length of the linear expressions is bounded, so in the Babelsberg/S implementation, we only initialize intermediate variables once, as the underlying array cannot grow or shrink. Since we only need to initialize them once, this can be taken care of by the framework.

Predicates over expressions are useful to state constraints on a collection as a whole, rather than on each of its elements. We have used this, for example, in our implementation of the Outside-Sum-Sudoku. Here, all elements in a collection must sum to the number outside the Sudoku. When one element changes, the others must change, too, to ensure the total sum does not.

We have found few use-cases for the most general collection-methods `do:`, `collect:`, and `select:` that could not be expressed using more specific methods. These methods create new collections from existing ones. What the developer means when using them and how to translate that meaning to the solver is less clear in the general case, and we do not support them for now. We have found that uses of `select:`, `collect:`, and `detect:` in predicate expressions can usually be replaced by the direct predicate methods. We have found that the iteration method `do:` is usually just used to express constraints on each element, and can usually be pulled outside of the constraint block. We might lift this restriction in the future if we find a significant number of uses of these methods in constraints that are much more expressive than their direct predicate counterparts. Until then, and for simplicity in the implementation, we do not allow these methods in constraints.

3.3 Maintaining Constraints in Babelsberg/S

Once asserted, constraints need to continue to be satisfied until they are disabled, all objects they apply to are garbage collected, or the program stops. To ensure this, the Babelsberg design follows the *perturbation* model established by the Kaleidoscope constraint-imperative language [14]. This model is similar to reactive systems in that changes to one part of the system propagate to other parts. In reactive systems, these changes are made by sampling a continuous process or through discrete events. In Babelsberg, the changes are the concrete event of assigning a new value to a variable that participates in a constraint. These changes then potentially propagate to other variables to keep constraints satisfied.

Each variable that participates in a constraint implicitly reacts to programmatic changes to its value by calling one or more solvers to re-satisfy the variable's constraints. Our wrappers around accessors intercept changes to variables that were used in a constraint and call `suggestValue:` on their associated `ConstraintVariable`. This adds a temporary equality constraint for the new value to the underlying constraint solver. The solver tries to solve all constraints. If the constraints are satisfiable, the new value is assigned. As a side-effect, other variables might change to satisfy constraints. If the solver cannot find a solution, a runtime error is generated and the new value is ignored.

In our Sudoku example, if a new value is assigned to a cell, the Sudoku constraints are solved in the background. If the solver finds a solution, the cell changes its value and the rest of the puzzle is adjusted to keep the Sudoku solveable. That is possible because the Sudoku application interacts with the underlying constraints.

Babelsberg/S can accommodate a variety of constraint solvers. Currently, it supports the Squeak implementation of Cassowary [1], and Z3 [4] through an IPC interface.

4. Evaluation

The constraints in Sudoku are easy to state, but not always easy to satisfy. A correct solution must assign each cell a number between 1 and 9 inclusively, while at the same time ensuring the no number occurs twice in a row, a column, or a block of 3 by 3 cells. We argue that logic puzzles such as Sudoku are good examples for interactive constraint applications. The user interface (written in Morphic) is shown in **Fig. 2**.

Listing 8 shows the constraints necessary to solve this Sudoku puzzle. These constraints use the Z3 constraint solver. Line 1 ensures that the user cannot change the numbers that were given initially. In some solvers, such as Cassowary, stay constraints can be used to express that the solver may not change a given variable, or to only change it if the constraints cannot be satisfied otherwise. Stay constraints are currently not supported in Z3, but will be in future versions. Currently, the method

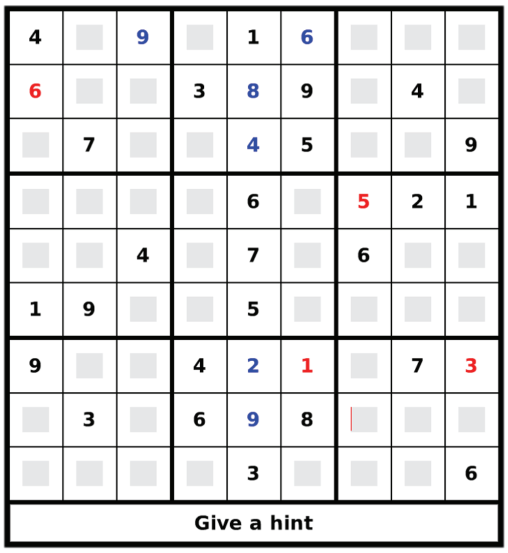


Fig. 2 The Morphic UI of the Sudoku puzzle. Some numbers (in black) are given initially. Each free cell allows the user to input a single number (in blue). The system can also generate hints (in red).

`addConstraintsForAllGivenNumbers` iterates over cells and creates a constraint that each cell that already has a value is always equal to just that value. Lines 3–4 assert the constraint that all cells must contain numbers between 1 and 9. Finally, lines 6–10 ensure that no row, column, or 3 × 3 box of cells can have duplicate numbers.

Note that the Squeak collection API does not contain a method `allDifferent`. Babelsberg/S adds this predicate for convenience. It is a normal object-oriented method in the `Collection` class that iterates over all elements in the collection and tests them for pairwise inequality. In ordinary code, this is just a test – the constraint interpreter, however, creates an inequality constraint expression for each comparison, exploding the allDifferent method into multiple constraints that the solver can understand. This means also, that subclasses can override the method and any different behavior will be reflected in the created constraints.

Note also that the normal accessor methods for rows and columns from Squeak `Matrix` objects are used, too. The Sudoku grid is just a subclass of `Matrix` that, besides a method to assert constraints on the given numbers, adds the `atBox:` accessor method to access each of the 9 boxes of size 3 × 3.

```
1  sudoku addConstraintsForAllGivenNumbers.
2
3  [sudoku allSatisfy: [:cell |
4    cell between: 1 and: 9]] alwaysSolveWith: solver.
5
6  (1 to: sudoku rowCount) do: [:index |
7      [(sudoku atRow: index) allDifferent &
8       (sudoku atColumn: index) allDifferent &
9       (sudoku atBox: index) allDifferent]
10         alwaysSolveWith: solver].
```

`Listing 8:` All constraints of a Sudoku puzzle.

As can be seen from Listing 8, the amount of code necessary for specifying all properties of a Sudoku puzzle is very small. With these, a solver can solve an arbitrary given Sudoku puzzle. The constraints are completely decoupled from the specific Sudoku puzzles and their given numbers.

In Babelsberg, constraints can be constructed, enabled and disabled at run-time, and, because they work correctly with method polymorphism, it is possible to subclass a logic puzzle to construct another by adding or removing constraints only. As an example, we have created Sudoku puzzle subclasses for *Diagonal Sudokus* and *Outside-Sum Sudokus*. In the former, the numbers of the two main diagonals have to be all different, and in the latter, the first three numbers in a row or a column must add up to a specific sum.

For a Diagonal Sudoku, provided there are accessor methods for the two diagonals, the method to create constraints is shown in Listing 9.

```
super createConstraints. "from normal Sudoku"
[(self diagonalFromTopLeft allDifferent)
  and: [self diagonalFromTopRight allDifferent]]
    alwaysSolveWith: solver.
```

`Listing 9:` The Diagonal Sudoku.

With object-constraint programming (OCP), it does not matter in which way a constraint variable or a constraint changes. The constraint satisfaction automatically works on each disturbance of the system. Currently, the values of cells only change when the user enters a new value into the morph that represents a cell. If that value is not a number between 1 and 9, or the Sudoku cannot be solved by adding this value, the solver rejects the input. However, the constraints encode no source for the change, so it does not matter if the change actually occurred through keyboard input. The Sudoku could also be calculated entirely by the computer, or the game could allow remote users to send values over the network. The constraints thus provide flexibility, because the developer does not need to know all events that might change the puzzle.

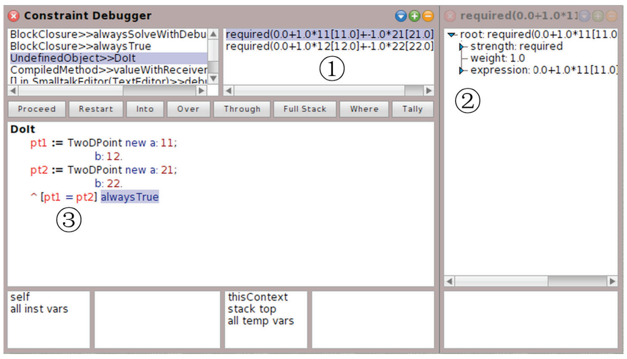
5. Conclusions

We have argued that OCP facilitates reactive systems in which dependencies between objects can be declared as constraints. It modularizes the relationship between objects and decouples constraint satisfaction from the application. Constraints can be dynamically added and removed, and are maintained automatically. This makes them useful for writing interactive applications. As an example, we implemented applications for specifying and solving different variants of Sudoku with constraints with a graphical user interface. The user can change the values of the constraint variables interactively without breaking the properties of the Sudoku. The application reacts on the user input by resolving the underlying constraints.

There are two major directions for future work. Regarding the implementation, we plan to implement an alternative solution to provide instance-specific wrappers. This will improve the compatibility of constrained objects with existing Smalltalk code. We also plan to support more features of the Babelsberg design as found in its JavaScript and Ruby implementations, such as incremental resolving, local propagation, and identity constraints.

Furthermore, we plan to leverage the Smalltalk metaprogramming facilities to explore how to aid developers in debugging and understanding constraints. If incorrect constraints are generated, why? If the solver cannot find a solution or is slow, what can be done? These are still open questions for Babelsberg, because constraints cannot be easily debugged.

Figure 3 shows how we extended the Squeak debugger to support stepping into constraints. Our debugger has an additional pane on the top right ①, and a special inspector on the right hand side ②. The debugger works as it normally would when running imperative code, but upon entering contraint construction mode,



```
BlockClosure>>alwaysSolveWithDebu
BlockClosure>>alwaysTrue
UndefinedObject>>DoIt
CompiledMethod>>valueWithReceiver
[] in SmalltalkEditor(TextEditor)>>debu
```

```
required(0.0+1.0*11[11.0]+-1.0*21[21.0]
required(0.0+1.0*12[12.0]+-1.0*22[22.0]
```

```
root: required(0.0+1.0*11[11.0
  strength: required
  weight: 1.0
  expression: 0.0+1.0*11[11.0]
```

```
DoIt
    pt1 := TwoDPoint new a: 11;
                b: 12.
    pt2 := TwoDPoint new a: 21;
                b: 22.
    ^ [pt1 = pt2] alwaysTrue
```

```
self
all inst vars
```

```
thisContext
stack top
all temp vars
```

Fig. 3 Squeak debugger with constraints.

the debugger additionally tracks the constraints as they are created. In the example, we assert that `pt1` and `pt2` should be equal ③. From just looking at the expression we cannot tell how many constraints would be created. We could infer from the implementation if the = method for 2d points that we will create two constraints, one for each pair of dimensions on those two points, but a debugger allows us to observe this fact and see the equations that have been translated for the Cassowary solver in the top right pane. On the right hand side, we can see the details of the first constraint and for example change its strength or the generated expression to see how that changes the program behavior. Additionally, we can step into the procedure that assigns updated values from the constraint solver to the program variables and thus see the global effects of a constraint. This is particularly useful to understand which solution a solver chooses for a particular constraints and how many variables are changed in which way. We plan to extend this prototype into a debugger that is useful to answer different questions that arise when developing with constraints.

Despite these avenues for future work, we think that Babelsberg/S is already a useful implementation of object-constraint programming and we plan to include it in a future release of the R/Squeak distribution, a Squeak distribution that includes research projects considered useful for general purpose development [*1].

References

[1] Badros, G.J., Borning, A. and Stuckey, P.J.: The Cassowary Linear Arithmetic Constraint Solving Algorithm, *ACM Trans. Computer-Human Interaction* (*TOCHI*), Vol.8, No.4, pp.267–306 (online), DOI: 10.1145/504704.504705 (2001).

[2] Borning, A., Freeman-Benson, B. and Wilson, M.: Constraint Hierarchies, *Lisp and Symbolic Computation*, Vol.5, No.3, pp.223–270 (online), DOI: 10.1007/bf01807506 (1992).

[3] Clarke, L.A.: A System to Generate Test Data and Symbolically Execute Programs, *IEEE Trans. Software Engineering*, Vol.2, No.3, pp.215–222 (online), DOI: 10.1109/tse.1976.233817 (1976).

[4] de Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*), pp.337–340, Springer (online), DOI: 10.1007/978-3-540-78800-3_24 (2008).

[5] Felgentreff, T., Borning, A. and Hirschfeld, R.: Specifying and Solving Constraints on Object Behavior, *Journal of Object Technology*, Vol.13, No.4, pp.1–38 (online), DOI: 10.5381/jot.2014.13.4.a1 (2014).

[6] Felgentreff, T., Borning, A., Hirschfeld, R., Lincke, J., Ohshima, Y., Freudenberg, B. and Krahn, R.: Babelsberg/JS, *Proc. European Conference on Object-oriented Programming* (*ECOOP*), Springer, pp.411–436 (online), DOI: 10.1007/978-3-662-44202-9_17 (2014).

[7] Felgentreff, T., Millstein, T.D., Borning, A. and Hirschfeld, R.: Checks and Balances: Constraint Solving Without Surprises in Object-Constraint Programming Languages, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), ACM, pp.767–782 (online), DOI: 10.1145/2814270.2814311 (2015).

[8] Freeman-Benson, B.N., Maloney, J. and Borning, A.: An Incremental Constraint Solver, *Communications of the ACM*, Vol.33, No.1, pp.54–63 (online), DOI: 10.1145/76372.77531 (1990).

[9] Graber, M., Felgentreff, T., Hirschfeld, R. and Borning, A.: Solving Interactive Logic Puzzles With Object-Constraints — An Experience Report Using Babelsberg/S for Squeak/Smalltalk, *Workshop on Reactive and Event-based Languages & Systems* (*REBLS*), pp.1:1–1:5 (2014).

[10] Hirschfeld, R., Costanza, P. and Haupt, M.: An Introduction to Context-Oriented Programming with ContextS, *Generative and Transformational Techniques in Software Engineering II*, Springer, pp.396–407 (online), DOI: 10.1007/978-3-540-88643-3_9 (2008).

[*1] https://www.hpi.de/swa/trac/SqueakCommunityProjects

[11] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S. and Kay, A.: Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), ACM, pp.318–326 (online), DOI: 10.1145/263698.263754 (1997).
[12] Ist, I.L., Lynce, I. and Ouaknine, J.: Sudoku as a SAT Problem, *Proceedings of the International Symposium on Artificial Intelligence and Mathematics* (*AIMATH*), Springer, pp.1–9 (online), DOI: 10.1.1.331.458 (2006).
[13] King, J.C.: Symbolic Execution and Program Testing, *Comm. ACM*, Vol.19, No.7, pp.385–394 (online), DOI: 10.1145/360248.360252 (1976).
[14] Lopez, G., Freeman-Benson, B. and Borning, A.: Kaleidoscope: A Constraint Imperative Programming Language, *Constraint Programming*, Springer, pp.313–329 (online), DOI: 10.1007/978-3-642-85983-0_12 (1994).
[15] Maloney, J.: *Morphic: The Self User Interface Framework*, 4th edition (1995).
[16] Maloney, J.: *An Introduction to Morphic: The Squeak User Interface Framework* (2001).
[17] Milicevic, A., Rayside, D., Yessenov, K. and Jackson, D.: Unifying Execution of Imperative and Declarative Code, *Proc. International Conference on Software Engineering* (*ICSE*), pp.511–520, ACM (online), DOI: 10.1145/1985793.1985863 (2011).
[18] Roy, P. and Pachet, F.: Reifying Constraint Satisfaction in Smalltalk, *Journal of Object-Oriented Programming*, Vol.10, No.4, pp.43–51 (1997).
[19] Sadun, E.: *iOS Auto Layout Demystified*, Addison-Wesley (2013).
[20] Taivalsaari, A., Mikkonen, T., Ingalls, D. and Palacz, K.: Web Browser as an Application Platform: The Lively Kernel Experience, Technical report, Sun Microsystems, Inc. (2008).
[21] Van Hentenryck, P., Michel, L., Perron, L. and Régin, J.-C.: Constraint programming in OPL, *Principles and Practice of Declarative Programming*, Springer, pp.98–116 (1999).

Appendix

A.1 Extension to the Formal Semantics of Babelsberg

The semantic rules presented here are an extension to the semantics of Babelsberg/Objects, presented in the companion technical report to [7]. This appendix should be read as an additional chapter after that companion report.

The syntax is augmented to include an element Ⓟ, which ranges over the core predicates on collections such as `allSatisfy:`, `anySatisfy:`, `includes:` and so on. For languages which support re-definition of the methods that come with the language, we assume that the element matches only the original implementations, not user-defined re-definitions. Furthermore, we augment the syntax to also support accessing records using expressions.

L-Value	`L`	`::=`	`x | e.l | e[e]`
Label	`l`	`::=`	record label names | Ⓟ

Table A·1 gives an overview of the additional judgments used in this extension of the semantics. We add two opaque helper judgments. The first converts constants to label names, and the second checks if a constant value refers to an array class type. Both are defined in terms of the host language API. Note that for languages the support re-definition of core classes, the second judgment will return `false` if such re-definition has taken place.

Besides those additions, we only add the extended evaluation rules for dynamic field access and the special constraint construction mode (*ccm*) for collection APIs. Note that we do not add a typing rule for dynamic field access — during inlining, such access are turned into ordinary field accesses, and their expressions are required to stay equal to the current value.

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e}_l\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\,|\,\mathbb{H}'\,|\,\mathbb{C}'\,|\,\mathbb{I}'\,|\,\texttt{c>} \qquad \mathit{asLabel}(\texttt{c}) = \texttt{l}\\ \texttt{<}\mathbb{E}'\,|\,\mathbb{S}'\,|\,\mathbb{H}'\,|\,\mathbb{C}'\,|\,\mathbb{I}'\,|\,\texttt{e.l>} \Downarrow \texttt{<}\mathbb{E}''\,|\,\mathbb{H}''\,|\,\mathbb{C}''\,|\,\mathbb{I}''\,|\,\texttt{v>}\end{array}}{\texttt{<}\mathbb{E}\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e[e}_l\texttt{]>} \Downarrow \texttt{<}\mathbb{E}''\,|\,\mathbb{H}''\,|\,\mathbb{C}''\,|\,\mathbb{I}''\,|\,\texttt{v>}}$$

(E-ExpField)

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_l\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_{C_l},\texttt{e}'_l\texttt{>}\\ \texttt{<}\mathbb{E}'\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e}_l\texttt{>} \Downarrow \texttt{<}\mathbb{E}''\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{c>} \qquad \mathit{asLabel}(\texttt{c}) = \texttt{l}\\ \texttt{<}\mathbb{E}'',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l>} \rightsquigarrow \texttt{<}\mathbb{E}''',\texttt{e}_C,\texttt{e}'\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e[e}_l\texttt{]>} \rightsquigarrow \texttt{<}\mathbb{E}''',\texttt{e}_{C_l}\wedge\texttt{e}_C\wedge\texttt{e}'_l\texttt{=c},\texttt{e}'\texttt{>}}$$

(I-ExpField)

We extend the inlining judgment to also work for statements. This is used in the inlining judgment to translate calls to the well-known collection predicates. These predicates will not match the previous I-MultiWayCall rule, because their implementations have more than a single return expression, so that rule is unchanged. Because we allow a limited subset of statements, including assignment to locals in inlining collection predicates, the inlining rule including statements also returns an updated scope. In addition, the constraint expressions that are returned by the inlining rule for statements are split into groups for conjunctions and disjunctions — this is required to track, based on the early returns that are encountered, whether of set of inlined expressions all just one needs to be satisfied.

We define a helper rule to inline collection predicates:

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}'\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e}_0\texttt{>} \Downarrow \texttt{<}\mathbb{E}''\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{v>}\\ \mathbb{E};\mathbb{H} \vdash \texttt{v : T} \qquad \mathit{isBasicCollection}(\texttt{T}) = \texttt{true}\\ \texttt{<}\mathbb{E}_0\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{v}_1\texttt{>}\\ \vdots\\ \texttt{<}\mathbb{E}_{n-1}\,|\,\mathbb{S}\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{e}_n\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\,|\,\mathbb{H}\,|\,\mathbb{C}\,|\,\mathbb{I}\,|\,\texttt{v}_n\texttt{>}\\ \texttt{e}_C = (\texttt{e=v} \ \wedge\ \texttt{e}_1\texttt{=v}_1 \ \wedge\ \cdots\ \wedge\ \texttt{e}_n\texttt{=v}_n)\\ \mathit{lookup}(\texttt{v,l}) = (\texttt{x}_1 \cdots \texttt{x}_n\texttt{,s; return c})\\ \mathit{enter}(\mathbb{E}_n,\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{v},\texttt{x}_1 \cdots \texttt{x}_n,\texttt{e}_1 \cdots \texttt{e}_n)\\ \qquad\qquad = (\mathbb{E}',\mathbb{S}_m,\mathbb{H},\mathbb{C},\mathbb{I})\end{array}}{\begin{array}{r}\mathit{preparePredicate}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l(e}_1\texttt{,...,e}_n\texttt{))}\\ = (\mathbb{E}',\mathbb{S}_m,\texttt{s; return c},\texttt{e}_C)\end{array}}$$

(PreparePredicate)

This rule sets up the required equalities for all the arguments and the receiver, and is essentially the same as I-Call. As an addition, it limits any inlining to collection predicates that return a constant as a final statement. In the two rules that follow, this constant is further limited to be either `true` or `false`.

$$\frac{\begin{array}{c}\mathit{preparePredicate}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.Ⓟ(e}_1\texttt{,...,e}_n\texttt{))}\\ \qquad = (\mathbb{E}',\mathbb{S}',\texttt{s; return c},\texttt{e}_0)\\ \texttt{c = true} \qquad \texttt{<}\mathbb{E}',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s>} \stackrel{\sim}{\rightsquigarrow} \texttt{<}\mathbb{E}'',\mathbb{S}',\texttt{e}_1,\texttt{e}_C,\texttt{e}_D\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.Ⓟ(e}_1\texttt{,...,e}_n\texttt{)>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_0\wedge\texttt{e}_1,\texttt{e}_C\texttt{>}}$$

(I-PositivePredicate)

Table A·1 Judgments and intuitions of additional and changed semantic rules.

Opaque Judgments	
$\mathit{asLabel}(\texttt{c}) = \texttt{l}$	Constant c converted into a label yields l
$\mathit{isBasicCollection}(\texttt{T}) = \texttt{c}$	When type T corresponds to a known basic collection type that is supported in constraints with predicates, c is true.
Constraint Solving	
$<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}> \rightsquigarrow <\mathbb{E}',\texttt{e}_0,\texttt{e}'>$	Inlining expression e in $\mathbb{S}$ is equivalent to e′ in $\mathbb{E}$ if $\texttt{e}_C$ evaluates to true.
$<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}',\mathbb{S}',\texttt{e}_0,\texttt{e}_c,\texttt{e}_d>$	Inlining statement s is equivalent to solving conjunction of constraint expressions $\texttt{e}_C$ and the disjunction of constraint expressions $\texttt{e}_D$ if $\texttt{e}_0$ evaluates to true. This inlining step returns an updated environment $\mathbb{E}'$ and scope $\mathbb{S}'$.
Helper Rule	
$\mathit{preparePredicate}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l}(\texttt{e}_1,\ldots,\texttt{e}_n)) = (\mathbb{E}',\mathbb{S}',\texttt{s; return c},\texttt{e}_C)$	Preparing the method call $\texttt{e.l}(\texttt{e}_1,\ldots,\texttt{e}_n)$ for inlining returns and updated environment $\mathbb{E}'$, the fresh method scope $\mathbb{S}'$, the method body s; return c, and is valid if $\texttt{e}_C$ evaluates to true.

$$\dfrac{\begin{array}{c}\mathit{preparePredicate}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.}\text{Ⓟ}(\texttt{e}_1,\ldots,\texttt{e}_n)) \\ = (\mathbb{E}',\mathbb{S}',\texttt{s; return c},\texttt{e}_0) \\ \texttt{c} = \texttt{false} \qquad <\mathbb{E}',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}'',\mathbb{S}',\texttt{e}_1,\texttt{e}_C,\texttt{e}_D>\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.}\text{Ⓟ}(\texttt{e}_1,\ldots,\texttt{e}_n)> \rightsquigarrow <\mathbb{E}'',\texttt{e}_0\wedge\texttt{e}_1,\texttt{e}_C \wedge \texttt{e}_D>}$$
(I-NegativePredicate)

We use two separate rules for inlining through collection predicates that return true or false as their final statement. For methods that return true any disjunction, which would be created by an early return true, does not have to be fulfilled, as even without the early return the method would return true. Conversely, when the method returns false, fulfilling any conjunction will not suffice, because that would simply prevent an early return false, but not the final return statement.

Since we now allow inlining through a limited subset of statements, we add inlining rules for those. Note that these rules can only come into play through an I-*Predicate. Furthermore, all rules not supplied here still lead to a failure to evaluate an I-*Predicate rule, and fall back to the previous I-Call rule to set up a one-way constraint on the result of the call.

$$\dfrac{\begin{array}{c}\mathbb{S}(\texttt{x})\uparrow \qquad \mathbb{E}(\texttt{x}_g)\uparrow \\ <\mathbb{E}|\mathbb{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\texttt{e}> \Downarrow <\mathbb{E}'|\mathbb{H}|\mathbb{C}|\mathbb{I}|\texttt{v}> \\ <\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}> \rightsquigarrow <\mathbb{E}'',\texttt{e}_0,\texttt{e}'> \\ \mathbb{S}' = \mathbb{S} \bigcup\{(\texttt{x},\texttt{x}_g)\} \qquad \mathbb{E}''' = \mathbb{E}'' \bigcup\{(\texttt{x}_g,\texttt{v})\} \\ \texttt{e}_c = (\texttt{e}_0 \wedge \texttt{e}' = \texttt{v} \wedge \texttt{x}_g = \texttt{v})\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{x := e}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}''',\mathbb{S}',\texttt{e}_c,\texttt{true},\texttt{false}>}$$
(I-AsgnNewLocal)

Assignments are only permitted to local variables. Since we can only start the statement inlining rules from a collection predicate Ⓟ, we start with a fresh scope and any local variable must be newly created first. In this case, assignment is turned into a required equality between the fresh variable name and the initial value. Note that we are using the expression judgment to evaluate the right-hand side, but we disallow any changes to the heap or the constraint stores.

$$\dfrac{\begin{array}{c}\mathbb{S}(\texttt{x}) = \texttt{x}_g \qquad \mathbb{E}(\texttt{x}'_g)\uparrow \\ <\mathbb{E}|\mathbb{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\texttt{e}> \Downarrow <\mathbb{E}'|\mathbb{H}|\mathbb{C}|\mathbb{I}|\texttt{v}> \\ <\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}> \rightsquigarrow <\mathbb{E}'',\texttt{e}_0,\texttt{e}'> \\ \mathbb{S}' = \mathbb{S} \setminus \{\texttt{x},\texttt{x}_g\} \qquad \mathbb{S}'' = \mathbb{S}' \bigcup\{(\texttt{x},\texttt{x}'_g)\} \\ \mathbb{E}''' = \mathbb{E}'' \bigcup\{(\texttt{x}'_g,\texttt{v})\} \\ \texttt{e}_c = (\texttt{e}_0 \wedge \texttt{e}' = \texttt{v} \wedge \texttt{x}'_g = \texttt{v})\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{x := e}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}''',\mathbb{S}'',\texttt{e}_c,\texttt{true},\texttt{false}>}$$
(I-AsgnLocal)

Since we do not allow creating additional constraints even in this extended inlining mode, there is no need to solve constraints when we re-assign to a local variable. Furthermore, since reassignments are needed for looping over collection indices, and these indices are also used to then access the collection, we create a fresh global name for every re-assigned variable. This way, every re-assignment turns into a new variable for the solver.

$$<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{skip}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E},\mathbb{S},\texttt{true},\texttt{true},\texttt{true}>$$
(I-Skip)

$$\dfrac{\begin{array}{c}<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_1> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}',\mathbb{S}',\texttt{e}_1,\texttt{e}_{C1},\texttt{e}_{D1}> \\ <\mathbb{E}',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_2> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}'',\mathbb{S}'',\texttt{e}_2,\texttt{e}_{C2},\texttt{e}_{D2}> \\ \texttt{e}_{C3} = \texttt{e}_{C1} \wedge \texttt{e}_{C2} \qquad \texttt{e}_{D3} = \texttt{e}_{D1} \vee \texttt{e}_{D2}\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_1\texttt{;}\texttt{s}_2> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E}'',\mathbb{S}'',\texttt{e}_1 \wedge \texttt{e}_2,\texttt{e}_{C3},\texttt{e}_{D3}>}$$
(I-Seq)

The skip and sequence rules are straightforward. The conjunction and disjunction expressions from the sequences are connected appropriately.

$$\dfrac{\begin{array}{c}\texttt{s = if e then return true else s}_1 \\ <\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}> \rightsquigarrow <\mathbb{E}',\texttt{e}_C,\texttt{e}'>\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E},\mathbb{S},\texttt{e}_C,\texttt{true},\texttt{e}'>}$$
(I-IfThenReturnTrue)

$$\dfrac{\begin{array}{c}\texttt{s = if e then return false else s}_1 \\ <\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}> \rightsquigarrow <\mathbb{E}',\texttt{e}_C,\texttt{e}'>\end{array}}{<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}> \mathrel{\substack{\rightsquigarrow \\ \rightsquigarrow}} <\mathbb{E},\mathbb{S},\texttt{e}_C,\texttt{e}'\texttt{=false},\texttt{false}>}$$
(I-IfThenReturnFalse)

We only support if-clauses used as early returns in this extended inlining mode. As described in Section 3, if the early return would return true, the inlined conditional is used in a disjunction, otherwise it is used in a conjunction.

$$\begin{array}{c}
\mathtt{s}_0 \ = \ \mathtt{while\ e\ do\ s} \\
<\mathbb{E}|\mathbb{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}> \Downarrow <\mathbb{E}'|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{true}> \\
<\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}> \rightsquigarrow <\mathbb{E}'',\mathtt{e}_0,\mathtt{e}'> \\
<\mathbb{E}'',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{s}> \stackrel{\rightsquigarrow}{\rightsquigarrow} <\mathbb{E}''',\mathbb{S}',\mathtt{e}_1,\mathtt{e}_{C_1},\mathtt{e}_{D_1}> \\
<\mathbb{E}''',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{s}_0> \stackrel{\rightsquigarrow}{\rightsquigarrow} <\mathbb{E}'''',\mathbb{S}'',\mathtt{e}_r,\mathtt{e}_{C_r},\mathtt{e}_{D_r}> \\
\mathtt{e}' \ = \ \mathtt{e}_0 \land \mathtt{e}' \land \mathtt{e}_1 \land \mathtt{e}_r \qquad \mathtt{e}_C \ = \ \mathtt{e}_{C_0} \land \ \mathtt{e}_{C_r} \\
\mathtt{e}_D \ = \ \mathtt{e}_{D_0} \lor \mathtt{e}_{D_r} \\
\hline
<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{s}_0> \stackrel{\rightsquigarrow}{\rightsquigarrow} <\mathbb{E}'''',\mathbb{S}'',\mathtt{e}',\mathtt{e}_C,\mathtt{e}_D>
\end{array}$$

(I-WhileDo)

$$\begin{array}{c}
\mathtt{s}_0 \ = \ \mathtt{while\ e\ do\ s} \\
<\mathbb{E}|\mathbb{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}> \Downarrow <\mathbb{E}'|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{false}> \\
<\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}> \rightsquigarrow <\mathbb{E}'',\mathtt{e}_0,\mathtt{e}'> \\
\hline
<\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{s}_0> \stackrel{\rightsquigarrow}{\rightsquigarrow} <\mathbb{E}'',\mathbb{S},\mathtt{e}_0 \land \mathtt{e}'\mathtt{=false},\mathtt{true},\mathtt{false}>
\end{array}$$

(I-WhileSkip)

Finally, the while construct is now supported during inlining. Note that the loop condition is inlined and required to stay at its value. This prevents the solver from being able to change the loop condition to, for example, satisfy the collection predicate only on a subset of the collection.

There is an issue with these rules: they may generate constraints that are too strong. Consider the following method:

```
def some_or_none()
  i := 0;
  while i < self.length do (
    if self[i] > 10 then return true;
    if self[i] < 0 then return false;
    i := i + 1
  );
  return true
end
always ary.some_or_none()
```

The constraint ensures that at least one element in the array is larger than ten, or else all elements are negative. Here, the constraint would be satisfied if:

$$\begin{array}{c}
\exists (x,i) \in \mathtt{ary}.\,(x > 10 \land (\forall (y,j) \in \mathtt{ary}.\neg(y < 0) \lor j > i)) \\
\bigvee \\
\forall (x,i) \in \mathtt{ary}.\neg(x < 0)
\end{array}$$

But the I-PositivePredicate rule would always require the conjunction to be satisfied, so the solver would have to solve this stronger constraints instead:

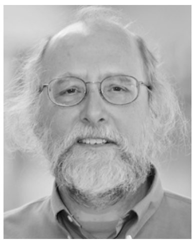
$$\forall (x,i) \in \mathtt{ary}.\neg(x < 0)$$

We have decided to avoid additional complexity in the rules to support generating the proper constraints in these cases. The code above could easily be rewritten to use two methods which each test one property, and then use these in a disjunction. Since the set of supported collection predicates Ⓟ is defined as part of the language, such methods may simply not be included in that set.

Tim Felgentreff is a Ph.D. student at the Software Architecture Group at the Hasso Plattner Institute (HPI), University of Potsdam, and a member of the HPI Research School for Service-Oriented Systems Engineering since 2013. His research interests are around programming language constructs and virtual machines. (See also http://www.hpi.de/swa/people/felgentreff)

Robert Hirschfeld is a professor of Computer Science at the Hasso Plattner Institute at the University of Potsdam, Germany. He is interested in improving the comprehension and design of software systems. Robert enjoys explorative programming in interactive environments. He served as a visiting professor at the Tokyo Institute of Technology and The University of Tokyo, Japan. Robert was a senior researcher with DoCoMo Euro-Labs, the European research facility of NTT DoCoMo Japan, where he worked on infrastructure components for next generation mobile communication systems with a focus on dynamic service adaptation and context-oriented programming. Prior to joining DoCoMo Euro-Labs, he was a principal engineer at Windward Solutions in Sunnyvale, California, where he designed and implemented distributed object systems, consulted in the area of object database technologies, and developed innovative software products and applications. Robert studied engineering cybernetics and computer science at Ilmenau University of Technology, Germany. (See also http://hpi.de/swa/)

Maria Graber has studied at the Hasso Plattner Institute until 2015. She received her masters degree at the Software Architecture Group for designing Babelsberg/S and extending it with constraints on collections. Maria now works at the IVU Traffic Technologies AG in Berlin, where she develops planning software for transportation companies.



Alan Borning is a professor in the Department of Computer Science & Engineering at the University of Washington, Seattle, USA, and an adjunct faculty member in the Information School. His research interests are in constraint-based languages and systems, object-oriented programming, human-computer interaction, and designing for human values. Current projects in programming languages include work on object constraint programming and constraint reactive programming languages, and in HCI, tools for making public transit more usable, and systems to support civic engagement and participation. He received a B.A. in mathematics from Reed College in 1971, and a Ph.D. in computer science from Stanford University in 1979. Awards include a Fulbright Senior Scholar Award for lecturing and research in Australia, and being named a Fellow of the Association for Computing Machinery in 2001.

Hidehiko Masuhara is a professor at the Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.Sc., M.Sc., and Ph.D. degrees from Department of Information Science, University of Tokyo in 1992, 1994, and 1999, respectively. His research interest is in programming languages and programming environments, especially advanced modularization mechanisms, optimization techniques, code recommendations, and debuggers.