Viewpoints Research Institute

Checks and Balances - Constraint Solving without Surprises in Object-Constraint Programming Languages

Tim Felgentreff, Todd Millstein, Alan Borning
and Robert Hirschfeld

VPRI Technical Report TR-2015-002

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

Checks and Balances

Constraint Solving without Surprises in Object-Constraint Programming Languages

Tim Felgentreff[1,2], Todd Millstein[1,3], Alan Borning[1,4], and Robert Hirschfeld[1,2]

[1]Communications Design Group (CDG), SAP Labs; Viewpoints Research Institute
[2]Hasso-Plattner-Institute, Potsdam, Germany; {firstname.lastname}@hpi.de
[3]University of California, Los Angeles, USA; todd@cs.ucla.edu
[4]University of Washington, Seattle, USA; borning@cs.washington.edu

Abstract

Object-constraint programming systems integrate declarative constraint solving with imperative, object-oriented languages, seamlessly providing the power of both paradigms. However, experience with object-constraint systems has shown that giving too much power to the constraint solver opens up the potential for solutions that are surprising and unintended as well as for complex interactions between constraints and imperative code. On the other hand, systems that overly limit the power of the solver, for example by disallowing constraints involving mutable objects, object identity, or polymorphic message sends, run the risk of excluding the core object-oriented features of the language from the constraint part, and consequently not being able to express declaratively a large set of interesting problem solutions.

In this paper we present design principles that tame the power of the constraint solver in object-constraint languages to avoid difficult corner cases and surprising solutions while retaining the key features of the approach, including constraints over mutable objects, constraints involving object identity, and constraints on the results of message sends. We present our solution concretely in the context of the Babelsberg object-constraint language framework, providing both an informal description of the resulting language and a formal semantics for a core subset of it. We validate the utility of this semantics with an executable version that allows us to run test programs and to verify that they provide the same results as existing implementations of Babelsberg in JavaScript, Ruby, and Smalltalk.

To appear, OOPSLA'15, Pittsburgh, Pennsylvania, 25-30 October 2015.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — constraints, classes and objects

Keywords Constraints, Object-Constraint Programming, Constraint Imperative Programming, Executable Semantics Test Suites

1. Introduction

Object-Constraint Programming (OCP) [4] integrates constraint satisfaction with common imperative and object-oriented features. It is motivated by the observation that a number of key aspects of interactive applications can be concisely specified using constraints, but that other aspects are more easily specified using standard imperative constructs (assignment, loops, and so forth). Its goal is to provide both paradigms in a cleanly integrated fashion, within an object-oriented framework that respects standard object-oriented features such as messages, encapsulation, and inheritance.

Experience with OCP systems has shown that natural solutions to a number of important problems involve constraints that include mutable objects, object identity, and message sends. However, such constraints also have the potential for surprising solutions and non-determinism. If a constraint references a field that does not exist on an object, should the solver be allowed to add it? If a constraint requires two variables to refer to the same object, which variable should be changed? Can methods be used in constraints? If so, can they only be used in the "forward" direction, to compute a result, or can the solver also use them to define multi-directional constraints?

In this paper we propose a set of design principles to answer these questions, along with a set of restrictions on the behavior of OCP systems that enforce these principles. The principles and restrictions are as follows:

- *Structure preservation*: Asserted constraints cannot change the structure of any objects (meaning the number and names of their fields). This property is enforced through a form of dynamic structural typechecking, along with implicitly generated extra constraints that implement frame axioms.

- *Identity preservation*: Similarly, newly asserted constraints cannot change what object a particular variable or field stores (such changes can only flow from assignments). This property is enforced by requiring constraints on object identity to already be satisfied at the point where they are asserted.
- *Structural/identity determinism*: The structure of objects as well as the particular objects stored by variables and fields must be allowed to change through imperative updates, and such changes can in turn cause existing constraints to be re-solved. However, the results of constraint solving are deterministic in terms of the final structures of objects and the identities of objects stored in each variable/field. This property is enforced through a novel two-phase solving process that first deterministically solves identity constraints and then solves the remaining constraints in the updated environment and heap.
- *Syntactic method rules*: A simple syntactic check suffices to determine whether and how a method can be used within constraints. In short, methods with side effects (writes other than to local variables) cannot be used in constraints. Other methods whose bodies simply compute and return a expression can be used *multi-directionally* in constraints — any subset of their variables can be considered "unknowns" and solved for in terms of the other variables — provided any methods called as part of that expression also follow this restriction. The remaining methods can be used just in the "forward" direction, in order to produce a result value which can participate within constraints.

We have instantiated our approach in the context of Babelsberg, a family of object-constraint languages. Instances of Babelsberg include Babelsberg/R (a Ruby extension) [4], Babelsberg/JS (a JavaScript extension) [5], and Babelsberg/S (a Squeak extension) [10]. However, the ideas are applicable to object-constraint languages in general. Indeed, Babelsberg inherits the described issues, as well as its overall approach, from the earlier Kaleidoscope [8, 16] and Turtle [11] constraint imperative programming languages, and our approach is also useful for constraint systems such as BackTalk [19], Kaplan [14], and Squander [17] (see Section 5).

We describe our approach informally (Section 2) and have evaluated it in several ways. First, we have formalized the approach in a core subset of Babelsberg (Section 3). Second, we have developed executable versions of the formal semantics of our core language as well as a full object-oriented language with polymorphic method dispatch. We use this executable semantics to automatically verify a suite of example programs, including ones that illustrate the problems we address. On top of our executable semantics, we have also built a framework to generate language test suites from our suite of example programs, to facilitate automatically testing Babelsberg implementations for conformance to the semantic rules. Finally, we have created an updated version of the Babelsberg/JS implementation using our approach, have verified that it conforms to the formal semantics on the suite of example programs, and describe our experience porting existing Babelsberg/JS applications to our new version (Section 4).

2. Overview

This section motivates the problems we address with object-constraint languages and presents our solutions informally, using Babelsberg to ground the discussion. Babelsberg allows constraints to be interleaved with imperative code and supports expressive constraints involving mutable objects and their methods. The language provides a mechanism to satisfy the constraints stated by the programmer, and to ensure that they remain satisfied when variables are re-assigned imperatively. A core principle of Babelsberg is that, in the absence of constraints, it should behave like a standard imperative language.

In Babelsberg, the details of the underlying solvers are abstracted so that constraints can be handed to a number of different solvers, including ones that can then cooperate to find solutions. For Babelsberg, the solver is used to find a single best solution — if there are multiple solutions, it is free to pick any one of them. Providing answers rather than solutions, i.e., results such as $10 \leq x \leq 20$ rather than a single value for x, and backtracking among multiple answers, as available in for example constraint logic programming [13], is not currently a goal.

Instances of Babelsberg are direct extensions of dynamic object-oriented languages, and include support for such features as dynamic typing, object encapsulation, and message sends in constraints. Previous work that integrated constraints into object-oriented languages did not unify these constructs for both paradigms, requiring programmers to essentially learn two different sets of abstraction mechanisms. As a result, modules must implement both object-oriented and constraint interfaces to be used in both paradigms, and constraint abstractions relied on direct access to object fields or special *constrainable* types, thus limiting the re-usability of constraint code across objects with similar interfaces, but different internal structure. OCP languages unify the abstraction mechanisms by using only object-oriented methods in both object-oriented code and constraints in a way that respects encapsulation and makes code re-use easier in semantically clean way.

Babelsberg implementations allow any object or value to occur in a constraint and (given capable solvers) be solved for—there is no distinction between *constrainable* variables and *ordinary* variables. However, during solving there is a distinction between objects for which identity is significant and those for which it is not. For example, a boolean or small integer is a true object in Ruby and Smalltalk, but its identity is determined purely by its value. Thus, variables referring to such objects can be re-assigned by the solver. Variables that refer to objects for which identity is significant, however, cannot simply be re-assigned by the solver—in Babelsberg, these may only be re-assigned by the solver to satisfy constraints over object identities in response to imperative assignments.

We will use Babelsberg/JS for the examples that follow (i.e., JavaScript syntax). Babelsberg/JS augments JavaScript with statements of the form `always: { e }` and `once: { e }` to declare constraints, where `e` is an ordinary JavaScript expression. In Babelsberg/JS, this expression is evaluated in a special interpreter mode to convert it dynamically into one or more constraints that can be handed to the solver and, when solved, will make the JavaScript expression return true. Both `always` and `once` constraints are solved immediately. An `always` constraint is required to hold for the rest of the execution; Babelsberg/JS therefore re-creates and re-solves the constraint as part of the execution of each subsequent statement if it may have become invalidated, which can happen whenever a variable or field is assigned. In contrast, a `once` constraint is solved once and then immediately retracted. Semantically, `always` constraints persist for the rest of the execution, but the practical implementations garbage collect constraints if all variables that they transitively refer to are no longer accessible.

As a simple example, consider a bank account application in which we want to prevent changes that would make the account balance drop below a certain threshold. Additionally, we want to track the daily interest, but not allow it to rise above 10 euros. Below is part of a Babelsberg/JS implementation of such an application.

Listing 1.

```
1 var acc = new Account(), plan = new SavingsPlan();
2 always: { acc.balance >= plan.minimumBalance()? }
3 always: { priority: "medium"
4               plan.dailyInterest == (acc.balance? * 0.01) / 365.0 }
5 always: { plan.dailyInterest <= 10 }
```

The constraint on line 2 ensures that `acc.balance` remains above the minimum balance required by the plan. Note that the plan's minimum balance is the result of a message send—it does not matter if it is calculated or a field access. If the constraint is already satisfied at that point, then nothing need be done; otherwise the solver will modify the values of variables and their fields in order to satisfy the constraint. Note also that these variables and fields are not special "constraint variables"—any variable can appear and potentially be solved for in a constraint. The read-only annotation on `plan.minimumBalance()`, indicated by the question mark, prevents the solver from changing the plan or its minimum balance to satisfy the constraint. The theory of read-only annotations applies only to single variables [2, 6] and we make the same restriction in our semantics below, but in practical languages we also permit read-only annotations on expressions. The implementations do a simple rewrite to convert a read-only expression to a read-only variable by introducing a fresh variable as needed [6].

In this constraint, the read-only annotation means that only `acc.balance` is available for the solver to modify. Because this is an `always` constraint, it will be re-checked, and if necessary re-solved, as part of executing each subsequent statement as well.[1] It is of course possible for the given constraints to be unsatisfiable. For example, if there is a required constraint that the minimum balance is 100, but a subsequent constraint requires that `acc.balance` be 50, or if that value is directly assigned to `acc.balance`, an exception will be generated.

Line 3 computes the daily interest and again uses a read-only annotation, now so that `acc.balance` cannot be changed to satisfy this constraint. By default, constraints have a priority of `required`, but Babelsberg/JS supports *soft constraints* as well. On line 3, the constraint priority is `medium`, which tells the system that it should satisfy the constraint if possible, but it is no error not to do so. Specifically, if the daily interest were to rise above 10, the system would choose to satisfy the constraint on line 4 but leave the constraint on line 3 unsatisfied. There are a number of alternatives for the definition of what constitutes an optimal solution given competing soft constraints, each with their own advantages and disadvantages [2, 6]. In the work here, we encode that definition in the choice of constraint solver, rather than making it a separate part of the semantic rules.

The rest of this section describes the problems that can arise with such an expressive object-constraint language, our design principles for addressing these problems, and our rules for enforcing the principles.

2.1 Structure Preservation

Consider again the bank account example shown above. The programmer expects the constraints to possibly affect the account's balance as well as the daily interest. However, in the absence of other restrictions, we discovered that Babelsberg, as well as earlier systems that integrate constraints and objects, would sometimes find unintuitive solutions that the programmer likely did not even consider, let alone intend.

Specifically, suppose the object `acc` has no `balance` field. In earlier versions of Babelsberg, the solver could silently invent such a field to satisfy the constraint on line 2, and that field would be added to the object when the solution is used to update the heap—after all, we did ask in our constraint that such a field exist and be equal to the minimum balance. (Similarly, if we created a constraint that the account be equal to an object with only a balance field

[1] The constraint is active from the point at which it is created and onwards, but does not retroactively apply prior to the execution. In a graphical application, for example, the user may have already seen previous results.

and if the account previously had additional fields, the superfluous fields could be removed.) While one might argue that such behavior is desirable in these cases, it is a slippery slope. For example, consider a constraint of the form `p.x > 0 || p.y > 0` on a point `p`. There is inherent nondeterminism here — the solver can choose which field to update if the constraint does not hold. But if `p` lacks an `x` field should it be invented? If `p` lacks both fields which one should be invented?

To avoid confusion, we employ a principle of *structure preservation*: the solution to a newly asserted constraint will never change the structure of any objects, where we take *structure* to include the names and number of an object's fields (and thus, implicitly also its size), recursively. We enforce this principle by employing a form of *structural type-checking* at run time, just before sending constraints to the solver. Our checks ensure that all structural requirements of the given constraints are already satisfied in the current environment and heap. For example, before solving the constraint added in line 2 of our bank account example, Babelsberg will check that `acc` already has a `balance` field and will raise an exception if that is not the case.

2.2 Identity Preservation

Another issue in the design of an object-constraint language is the interaction between constraints and object identity. Object identity plays an essential role when using an object-oriented language to model aspects of the real world and so, in a language that integrates constraints with object-oriented features, we need to resolve the tension between these fundamental features [15].

Surprisingly, this tension arises even in the simple bank account example shown earlier. Consider the constraint on `acc.balance` in line 2. While the programmer expects the value of the account's `balance` field to be updated as necessary to satisfy the constraint, earlier versions of Babelsberg and prior object-constraint languages could also choose to simply change `acc` to point to another object whose `balance` field satisfies the constraint!

Additional complications arise in the presence of explicit constraints on object identity. Such constraints are useful for specifying real-world requirements that two variables denote the same actual object and to describe cyclic structures. Kaleidoscope had many of the same goals as Babelsberg, as well as some of its features (including explicit constraints on object identity), but did not have the rules presented here to tame the power of the constraint solver. Experience with that language in particular demonstrated that identity constraints can have non-obvious consequences in an imperative language. As an example, suppose we add the following constraint after Listing 1:

Listing 2.

```
always: { acc === acc2 }
```

This identity constraint requires two variables to point to the same object. Here it is not clear whether the solution will require them both to point to the object stored in `acc` or that stored in `acc2`. Further, if these objects have different structures, this could lead to nondeterministic failures in subsequent structural checks. Finally, the solver could also satisfy the identity constraint by assigning both variables to yet a third object[2]. Adding a read-only annotation on one of the variables does not suffice, because that would prevent the system from re-satisfying the constraint if we later assign to the variable that is not read-only.

To address these problems, we employ a principle of *identity preservation*: newly asserted constraints cannot change which object a particular variable or field stores. To enforce this principle, we require all identity constraints to be *already satisfied* at the time they are asserted. Only during later assignments will the system re-satisfy the constraint deterministically. In our example above, the programmer must explicitly resolve the nondeterminism, for example by assigning `acc2` to point to the value of `acc`, before asserting the identity constraint. If this check fails we treat the constraint as unsatisfiable. While this approach places a bit more burden on the programmer, we believe that the programmer effort is more than made up for by the strong guarantee provided by the language.

2.3 Structural/Identity Determinism

Of course, imperative programs do need to sometimes update the structures of objects and change which objects are stored in particular variables and fields. In Babelsberg we must therefore allow such updates. The challenge, then, is to provide this expressiveness while at the same time avoiding the kinds of surprising behaviors described earlier when attempting to re-establish violated constraints.

To that end, we allow arbitrary updates to objects and variables through ordinary assignment but impose a principle of *structural and identity determinism*: all solutions to the violated constraints must agree on the structures of all objects and the identities stored in all variables and fields. The key observation is that changes to structure and identity can only occur through an assignment statement, and the inherent directionality of an assignment ensures a deterministic "flow" to other variables in order to re-establish violated constraints.

However, making this approach work requires a few new restrictions, particularly with respect to identity constraints. Specifically, Babelsberg requires identity constraints to be asserted on their own, separate from other constraints, and it forbids such constraints from appearing within disjunctions or negations. To see why this approach ensures determinism

[2] One could also imagine an *isDistinct* identity constraint to express that two variables should not refer to the same object—however, this would almost always lead to non-determinism, as the solver would be free to pick any object to re-satisfy the constraint if one of the variables were assigned to be identical to the other.

consider again the constraint in Listing 2 and suppose that acc2 is subsequently re-assigned to point to some object o. Since that re-assignment requires acc2 to point to o, the only way to re-satisfy the existing identity constraint is to re-assign acc to point to o as well.

Finally, an assignment statement may cause violations in both identity constraints as well as ordinary "value" constraints. As described in earlier subsections, constraint solving is normally not allowed to modify the structure of objects or change the objects stored in variables and fields. To re-establish all constraints in a manner that adheres to our restrictions and is understandable for programmers, we therefore introduce a two-phase approach to constraint solving as part of an assignment statement. In the first phase, all identity constraints are re-established as described above. This phase may update the structure of objects and change which objects are stored in particular variables and fields but will do so deterministically. In the second phase, all other constraints are solved in the context of this updated environment and heap, using the restrictions described earlier. Here we also include a notion from previous system [8] that the solver should always pick a solution that is "close" to the current state of the system. To that end, we implicitly add a low priority "stay" constraint to each variable, so that its value only changes if required by some higher-priority constraint.

As an example for our two-phase solving, suppose we have the constraints from Listing 1 plus the identity constraint in Listing 2, and we now assign an object {credit: 10} to acc2. In the first phase, we re-satisfy the identity constraints by propagating this assignment to acc. Now both again point to the same object. In the second phase, we attempt to typecheck and re-satisfy the value constraints on acc. However, these now attempt to constrain the balance field on an object with only a credit field—the structural typecheck thus fails, and we cannot allow the assignment as we could not re-satisfy the value constraints with the new value. The practical implementations of Babelsberg undo the effects of the first phase and generate a runtime exception, leaving all variables and fields unchanged. In the semantics, we say that the execution stops in this case.

2.4 Invoking Methods in Constraints

Just as procedural abstraction is very useful in imperative code, object-constraint languages require a way to name and parameterize constraints. Prior languages such as Kaleidoscope and Turtle accomplish this by introducing a new abstraction that is the analogue of procedures for constraints. This approach is semantically simple but makes the language larger, requires programmers to carefully separate their imperative and declarative code, and requires duplication for functionality that can be used both imperatively and declaratively.

Babelsberg instead unifies these abstractions, allowing constraints to invoke ordinary methods directly. This approach simplifies programming by avoiding the problems described above. However, it also creates new semantic challenges. The Babelsberg implementation must be able to translate method bodies into constraints. However, not all imperative code can be translated to declarative constraints. In earlier versions of Babelsberg, the line between methods that could successfully be called from constraints and those that could not was not precise, and evolved alongside the implementations in an *ad hoc* manner.

In this work we provide a simple set of syntactic criteria that allows programmers to easily understand whether and how a given method can be called from a constraint. In the rest of this section we discuss several semantic issues and how they are addressed. The resulting rules are more restrictive than those for the prior Babelsberg implementations, but they still allow a wide range of useful programs while remaining easily understood.

2.4.1 Dynamic Dispatch

An immediate concern with allowing methods to be invoked in constraints is the fact that methods are dynamically dispatched: different method implementations can be invoked at a call site, depending on the run-time class of the receiver object. Thanks to our restrictions on object identities along with the two-phase constraint-solving process described earlier, this turns out to be a non-issue. Specifically, after the first phase of constraint solving, all object identities are fixed. At that point, the dynamic dispatch can be performed in order to identify the appropriate method implementation, which can then be translated to a constraint as part of the second, "value" phase of constraint solving. The only new restriction we require to make this approach work is that methods called from constraints cannot include identity constraints, as these would have needed to be solved in the first phase.

2.4.2 Side Effects

Methods with side effects cannot safely be used in constraints. The presence of such side effects could cause the program's behavior to be dependent on the number of times that a constraint is checked, making programs very difficult to reason about. Side effects in constraints could also cause other constraints to become violated, leading to semantic challenges such as the potential for the constraint solving process to diverge, or could lead to situations where a constraint is seemingly satisfied, but would be unsatisfied at a later time without any of the participating variables changing (e.g., when reading from a file that was deleted outside the program).

We use a simple syntactic rule to ensure that methods used in constraints have no side effects. Such methods can only write to local variables and can only invoke other side-effect-free methods. This rule is checked dynamically before constraint solving by executing the constraint expression and detecting invalid operations.

However, we would also like to allow the creation of new objects in constraint expressions. Doing so is problematic if the identity of the newly created object could be significant, for example via reflection or explicit identity tests, yet we found numerous useful examples that involved creating simple objects such as points and rectangles in constraint expressions. To resolve this issue, we add the concept of *value classes* to Babelsberg and allow instances of these classes to be created in constraints. Unlike regular objects, instances of value classes are immutable after creation, and they do not have object identity. However, value classes are more than simple records, since they support methods and inheritance. A number of existing languages support or have proposals for forms of value classes, for example Scala[3] and Java[4].

2.4.3 One-Way and Multi-Way Constraints

An important advantage of constraints is that in general they are *multi-way*: any subset of their variables can be considered "unknowns" and solved for in terms of the other variables. In contrast, imperative code typically computes from a pre-determined set of inputs to a pre-determined output. In general, imperative code is not "reversible" for many reasons, including complex arithmetic, recursion, and unbounded loops. To correctly look up methods in constraints, as described in Section 2.4.1, we also need to determine the identities of variables encountered in value constraints. For chained method calls, this includes determining the identity of method return values, which, in the presence of branching and loops, would entail considering the possible control flow paths. Prior versions of Babelsberg used a kind of best-effort strategy, which made it difficult to know what would work when. This presented debugging challenges, because if the system rejected a constraint it was not always clear whether this was because it was incorrect or whether the system was unable transform control flow structures to determine the identities of the returned objects.

Our current rules instead rely on simple syntactic properties of the methods called in a constraint. If a method consists solely of a single `return` statement, then it can be used in a constraint in a multi-way manner, provided any methods called as part of that statement (recursively) themselves follow this restriction as well. While seemingly restrictive, this approach supports a spectrum of useful programming idioms. For example, suppose that we want to express the requirement that a window in a graphical application stay centered on the mouse position, which we keep in a variable `mousePosition`. This constraint can then be defined as follows:

Listing 3.

```
var window = new Window(0,0,100,100),
    mousePosition = $world.hand().getPosition();
always: { window.getCenter().equals(mousePosition) }
```

After running this code, the system will ensure that the window is centered on the value of `mousePosition`. If the variable is updated (e.g., through a mechanism that fires when the cursor moves) the system will adjust other parts of the window to keep the center at the desired position. If the window moves by some other means, the system changes the `mousePosition` variable to follow. If an adjustment is not possible, the system prevents the interaction.

In this example, it does not matter if `getCenter` is just an accessor or a calculated property; in either case, Babelsberg traces into the method and creates the correct relations between the actual parts of the window and the point. For example, the following implementation of `getCenter` is allowed by our rules:

Listing 4.

```
function getCenter() {
    return Point(
        this.topLeft.x + this.width / 2,
        this.topLeft.y + this.height / 2
    );
}
```

Because the method consists of only a `return` statement, Babelsberg can solve the constraint involving this method multi-directionally; that is, it can modify any or all of the `topLeft`, `width`, and `height` fields of the window in order to satisfy the constraint whenever the `mousePosition` changes.

A method `m` that is used in a multi-way constraint in the above manner is ultimately exploded into a set of primitive constraints that correspond to the primitive operations performed by `m` and its callees, which are then handed to the solver. However, if the resulting constraints are not supported by the solver, then even though we hand it to the solver as a multi-way constraint, the solver may be unable to solve for all variables that occur in the constraint or fail to solve the constraint altogether. For example, a local propagation solver such as DeltaBlue can handle the integer addition constraints resulting from the `getCenter` example but cannot handle a method that involves integer inequalities. (Given a constraint $x \leq y$, a local propagation solver cannot find a unique value for y given a value for x.) On the other hand, a solver that can accommodate both linear equalities and inequalities *would* be able to solve such a set of constraints. Such possible limitations of the solver are a separate issue from the rules given here. For example, given a constraint that invokes a method `encrypt` on a public key object, presumably no practical solver will be able to solve this in a multi-way manner in reasonable time—but if such

[3] `http://docs.scala-lang.org/sips/completed/value-classes.html`

[4] `http://openjdk.java.net/jeps/169`

a solver were ever developed, our rules would allow it to be used.

Given a side-effect-free method, if that method has branches or multiple statements rather than a single `return` statement, then we do not support using it in a multi-way manner. (This *will* prevent the system from reversing most `encrypt` methods, since they usually involve branches and loops even though they are side-effect-free.) The method *can* still be invoked within a constraint, but can only be used in the "forward" direction. That is, the method will simply be executed (rather than being exploded into further constraints), and its result value can be used to constrain other variables in the constraint. This may be useful, for example, if we just want to track the center of the window in a variable to display on the screen, and not allow changes to the variable that would affect the window. Using methods in this way is safe as long as there are no cyclic dependencies (e.g., the result of a method call constraining one of its inputs), which is checked during translation to the solver.

It would be possible to make our rules somewhat more lenient; for example, it would be possible to translate the following version of `getCenter` so that it can be used in a multi-way manner.

Listing 5.

```
function getCenter() {
    var center = // ... previous code
    if (this.isScaled) {
        center.x = center.x / this.scale;
        center.y = center.y / this.scale;
    }
    return center;
}
```

However, in practice we have found that as methods become more complex than a single expression, translating them can lead to surprising solutions. Even in this simple example, it is difficult to judge if the solver might also modify the `isScaled` or `scale` fields to satisfy the constraint, and what effects that might have. Further, as the methods become more complex, even if they are still side-effect-free, it quickly becomes impractical (and in the limit impossible) to reverse them. Our experience to date thus suggests that the current rule, while restrictive, provides a useful balance between clarity and power.

3. Formalism

This section presents a core language that we call Babelsberg/UID, which we use to formalize the key aspects of our approach. Specifically, the goals of Babelsberg/UID are to elucidate the relationship between imperative code and constraints, and to make precise our rules for taming this relationship as described in Sections 2.1-2.3 above. The formalism lacks methods and therefore does not account for our rules on method calls as discussed in Section 2.4. It also omits many other features that would be inherited from a host language, such exception handling and I/O.

In addition to an environment and heap, the semantics also maintains a current set of constraints, which arise from `always` statements. When a new constraint is asserted via `once` or `always`, that constraint along with the current set of constraints is passed to the constraint solver. If the solver finds a solution, it returns a new environment and heap to replace the existing ones; otherwise execution halts. In practical languages, a run-time exception would be generated and the heap and environment remain unchanged. An assignment statement is also modeled as a constraint: we evaluate the right-hand side and then create a constraint that the resulting value and the left-hand side be identical. This constraint is then used as part of constraint solving along with the current set of constraints to produce the updated environment and heap.

3.1 Syntax

We use the following syntax for Babelsberg/UID:

```
Statement       s  ::=  skip | s;s
                     |  L := e | x := new o
                     |  always C | once C
                     |  if e then s else s
                     |  while e do s
Constraint      C  ::=  ρ e | C ∧ C
Expression      e  ::=  v | L | e ⊕ e | 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
Value           v  ::=  c | r
```

Metavariable `c` ranges over the `nil` value, booleans, and primitive type constants. A finite set of operators on primitives is ranged over by $\oplus$. We assume $\oplus$ includes at least an equality operator = for each primitive type and an operator $\wedge$ for boolean conjunction. The operator == tests for identity — for primitive values this behaves the same as =. The symbol ρ ranges over constraint *priorities* and is assumed to include a bottom element `weak` and a top element `required`. The syntax requires the priority to be explicit; for simplicity we sometimes omit it in the rules and assume a priority of `required`.

3.2 Operational Semantics

The semantics includes an environment `E` and a heap `H`. The former is a function that maps variable names to values, while the latter is a function that maps mutable references to "objects" of the form $\{l_1{:}v_1,\ldots,l_n{:}v_n\}$. When convenient, we also treat `E` and `H` as sets of pairs (`{(x,v),...}` and `{(r,o),...}`, respectively). The currently active value con-

Expression Evaluation	
$\mathtt{E;H} \vdash \mathtt{e} \Downarrow \mathtt{v}$	Expression e evaluates to value v in the context of environment E and heap H
Typechecking	
$\mathtt{E;H} \vdash \mathtt{e} : \mathtt{T}$	Expression e has type T in the context of environment E and heap H
$\mathtt{E;H} \vdash \mathtt{C}$	Constraint C is well formed in the context of environment E and heap H
Constraint Solving	
$\mathtt{E;H} \models \mathtt{C}$	Environment E and heap H represent a solution to constraint C
Statement Evaluation	
$\mathtt{<E\|H\|C\|I\|s>} \longrightarrow \mathtt{<E'\|H'\|C'\|I'>}$	Execution starting from configuration <E|H|C|I|s> ends in the state <E'|H'|C'|I'>
Helper Rules	
$\text{solve}(\mathtt{E}, \mathtt{H}, \mathtt{C}, \rho) = \mathtt{E';H'}$	Solving the constraint C by translating it into a form suitable for the solver and adding stay constraints on E and H yields new environment E' and new heap H'
$\mathtt{E;H} \vdash \mathtt{C} \rightsquigarrow \mathtt{C'}$	Translating constraint C into a form suitable for the solver yields constraint C'
$\text{stay}(\mathtt{E}, \rho) = \mathtt{C}$	Constraint C is a conjunction of stay constraints on variables in environment E
$\text{stay}(\mathtt{H}, \rho) = \mathtt{C}$	Constraint C is a conjunction of stay constraints on objects in heap H
$\text{stay}(\mathtt{x=c}, \rho) = \mathtt{C}$	Constraint C is a weak stay constraint for x to equal c
$\text{stay}(\mathtt{x=r}, \rho) = \mathtt{C}$	Constraint C is a stay constraint with priority ρ for x to equal r
$\text{stay}(\mathtt{r=o}, \rho) = \mathtt{C}$	Constraint C is a required stay constraint for reference r to refer to the object o
$\text{stayPrefix}(\mathtt{E}, \mathtt{H}, \mathtt{I}) = \mathtt{C}$	Constraint C is a conjunction of required equalities to ensure both sides of the identity constraint can only be updated deterministically
$\text{stayPrefix}(\mathtt{E}, \mathtt{H}, \mathtt{L}) = \mathtt{C}$	Constraint C is a conjunction of required equalities to ensure that all but the last part in L cannot be changed

Table 1. Judgments and Intuitions of Semantic Rules

straints are kept as a compound constraint C; identity constraints are kept as a compound constraint referred to as I. Table 1 summarizes the judgments in our semantics and their intuitions. We give the rules to these judgments below.

Expression Evaluation The rules for evaluation are standard, but we show them for completeness. For each operator ⊕ in the language we assume the existence of a corresponding semantic function denoted $[\![\oplus]\!]$.

$$\mathtt{E;H} \vdash \mathtt{c} \Downarrow \mathtt{c} \qquad \text{(E-CONST)}$$

$$\mathtt{E;H} \vdash \mathtt{r} \Downarrow \mathtt{r} \qquad \text{(E-REF)}$$

$$\frac{\mathtt{E(x) = v}}{\mathtt{E;H} \vdash \mathtt{x} \Downarrow \mathtt{v}} \qquad \text{(E-VAR)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{r} \quad \mathtt{H(r)} = \{\mathtt{l}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v}_n\} \quad 1 \leq i \leq n}{\mathtt{E;H} \vdash \mathtt{L.l}_i \Downarrow \mathtt{v}_i} \qquad \text{(E-FIELD)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{e}_1 \Downarrow \mathtt{v}_1 \quad \mathtt{E;H} \vdash \mathtt{e}_2 \Downarrow \mathtt{v}_2 \quad \mathtt{v}_1 \ [\![\oplus]\!] \ \mathtt{v}_2 = \mathtt{v}}{\mathtt{E;H} \vdash \mathtt{e}_1 \ \oplus \ \mathtt{e}_2 \Downarrow \mathtt{v}} \qquad \text{(E-OP)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{L}_1 \Downarrow \mathtt{v} \quad \mathtt{E;H} \vdash \mathtt{L}_2 \Downarrow \mathtt{v}}{\mathtt{E;H} \vdash \mathtt{L}_1 \ \mathtt{==} \ \mathtt{L}_2 \Downarrow \mathtt{true}} \qquad \text{(E-IDENTITYTRUE)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{L}_1 \Downarrow \mathtt{v}_1 \quad \mathtt{E;H} \vdash \mathtt{L}_2 \Downarrow \mathtt{v}_2 \quad \mathtt{v}_1 \neq \mathtt{v}_2}{\mathtt{E;H} \vdash \mathtt{L}_1 \ \mathtt{==} \ \mathtt{L}_2 \Downarrow \mathtt{false}} \qquad \text{(E-IDENTITYFALSE)}$$

Typechecking We use a notion of typechecking to prevent undesirable non-determinism in constraints. Specifically, we want constraint solving to preserve the structure of the values of variables, changing only the underlying primitive data as part of a solution, in support of the goals listed in Sections 2.1 through 2.3. We formalize our notion of structure through a simple syntax of types:

$$\text{Type } \mathtt{T} \ \mathtt{::=} \ \mathtt{PrimitiveType} \mid \{\mathtt{l}_1\mathtt{:T}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:T}_n\}$$

Our typechecking rules check expressions dynamically just before constraint solving, so we typecheck in the context of a run-time environment. Otherwise the rules are relatively standard. In a statically typed language these checks could instead be performed in a more traditional manner (using the static types) at compile time. Note that we do not include type rules for identities. This ensures that constraints involving them do not typecheck, so identity checks cannot occur in ordinary constraints.

$$\mathtt{E;H} \vdash \mathtt{c} : \mathtt{PrimitiveType} \qquad \text{(T-CONST)}$$

$$\frac{\begin{array}{c}\mathtt{H(r)} = \{\mathtt{l}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v}_n\} \\ \mathtt{E;H} \vdash \mathtt{v}_1 : \mathtt{T}_1 \cdots \mathtt{E;H} \vdash \mathtt{v}_n : \mathtt{T}_n\end{array}}{\mathtt{E;H} \vdash \mathtt{r} : \{\mathtt{l}_1\mathtt{:T}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:T}_n\}} \qquad \text{(T-REF)}$$

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

$$\frac{\mathtt{E;H}\vdash\mathtt{L}:\{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}\quad 1\le i\le n}{\mathtt{E;H}\vdash\mathtt{L.l}_i:\mathtt{T}_i}\qquad\text{(T-FIELD)}$$

$$\frac{\mathtt{E;H}\vdash\mathtt{e}_1:\mathtt{PrimitiveType}\quad \mathtt{E;H}\vdash\mathtt{e}_2:\mathtt{PrimitiveType}}{\mathtt{E;H}\vdash\mathtt{e}_1\ \oplus\ \mathtt{e}_2:\mathtt{PrimitiveType}}\qquad\text{(T-OP)}$$

The following two rules define a constraint to be well-formed if it has a type.

$$\frac{\mathtt{E;H}\vdash\mathtt{e}:\mathtt{T}}{\mathtt{E;H}\vdash\rho\ \mathtt{e}}\qquad\text{(T-PRIORITY)}$$

$$\frac{\mathtt{E;H}\vdash\mathtt{C}_1\quad \mathtt{E;H}\vdash\mathtt{C}_2}{\mathtt{E;H}\vdash\mathtt{C}_1\ \wedge\ \mathtt{C}_2}\qquad\text{(T-CONJUNCTION)}$$

Constraint Solving This judgment represents a call to the constraint solver, which we treat as a black box. We assume that the solver supports our primitive types, records, and uninterpreted functions, as well as hard and soft constraints [2]. The proposition $\mathtt{E;H}\models\mathtt{C}$ denotes that environment E and heap H are an *optimal solution* to the constraint C, according to the solver's semantics. If there are multiple optimal solutions, the solver is free to pick any one of them. (As noted previously, we encode the precise definition of what constitutes an optimal solution in the choice of constraint solver, rather than making it a separate part of the semantic rules.)

Statement Evaluation A "configuration" defining the state of an execution includes a concrete context, represented by the environment and heap, a symbolic context, represented by the value and identity constraint stores, and a statement to be executed. The environment, heap, and statement are standard, while the constraint stores are not part of the state of a computation in most languages. Intuitively, the environment and heap come from constraint solving during the evaluation of the immediately preceding statement, and the constraints C and I record the (respectively, value and identity) always constraints that have been declared so far during execution. Note that our execution implicitly gets stuck if the solver cannot produce a model.

The first two rules below provide the semantics of value constraints. We require the constraint $\mathtt{C}_0$ to typecheck, which ensures that the structures of objects cannot change as a result of constraint solving. This also ensures that the given constraint contains no identity tests, which are handled separately.

$$\frac{\mathtt{E;H}\vdash\mathtt{C}_0\quad \text{solve}(\mathtt{E},\mathtt{H},\mathtt{C}\wedge\mathtt{C}_0,\mathtt{required})=\mathtt{E}';\mathtt{H}'}{\mathtt{<E|H|C|I|once\ C_0>}\longrightarrow\mathtt{<E'|H'|C|I>}}\qquad\text{(S-ONCE)}$$

$$\frac{\mathtt{<E|H|C|I|once\ C_0>}\longrightarrow\mathtt{<E'|H'|C|I>}\quad \mathtt{C}'=\mathtt{C}\wedge\mathtt{C}_0}{\mathtt{<E|H|C|I|always\ C_0>}\longrightarrow\mathtt{<E'|H'|C'|I>}}\qquad\text{(S-ALWAYS)}$$

Rule S-ONCE employs a helper judgment to perform constraint solving, which is used both for solving value and identity constraints and is defined by rule SOLVE in Figure 1. The rule generates "stay" constraints on the environment and heap, translates constraints to the language of the solver, and solves the constraints. The generation of stay constraints is defined by the remaining rules in the figure. Each variable and field has a stay constraint to keep it at its current value. Rule STAYCONST uses a weak priority to allow a primitive to be changed by the solver. Rule STAYOBJECT uses a required constraint on the structure of each object, to ensure that the solver will not invent new fields. In that rule we use H as an uninterpreted function to properly model the heap as a set of constraints. All other stay constraints have the given priority ρ. In the context of S-ONCE ρ is required, which prevents the structure of objects and the pointer relation among object references from changing when value constraints are solved.

Finally, the judgment $\mathtt{E;H}\vdash\mathtt{C}\rightsquigarrow\mathtt{C}'$ translates constraints into a form suitable for the solver. Specifically, each occurrence of an expression of the form L.l, where L refers to a heap reference r, is translated into H(L).l (recursively, as required), and each identity operator == is translated to the = operator. These rules are straightforward and are omitted.

The next two rules describe the semantics of identity constraints. The rules simply require that an identity constraint is already satisfied when it is asserted; hence the environment and heap are unchanged.

$$\frac{\mathtt{E;H}\vdash\mathtt{L}_0\Downarrow\mathtt{v}\quad \mathtt{E;H}\vdash\mathtt{L}_1\Downarrow\mathtt{v}}{\mathtt{<E|H|C|I|once\ L_0\ ==\ L_1>}\longrightarrow\mathtt{<E|H|C|I>}}\qquad\text{(S-ONCEIDENTITY)}$$

$$\frac{\begin{array}{c}\mathtt{<E|H|C|I|once\ L_0\ ==\ L_1>}\longrightarrow\mathtt{<E|H|C|I>}\\ \mathtt{I}'=\mathtt{I}\wedge\mathtt{L}_0==\mathtt{L}_1\end{array}}{\mathtt{<E|H|C|I|always\ L_0\ ==\ L_1>}\longrightarrow\mathtt{<E|H|C|I'>}}\qquad\text{(S-ALWAYSIDENTITY)}$$

The rule below describes the semantics of assignments. We employ the two-phase process described in the previous section, each represented by a "solve" premise. First the identity constraints are solved in the context of the new assignment. This phase only asserts weak stay constraints, thereby allowing the effect of the assignment to propagate through the identities, potentially changing the structures of objects as well as the relationships among objects in the environment and heap. In the second phase, the value constraints are typechecked against the new environment and heap resulting from the first phase. If they are well typed, then we proceed to solve them in the same manner as a once constraint (see S-ONCE above). This phase can change the values of primitives but will not modify the structure of any object.

$$\frac{\begin{array}{c}\text{stay}(\texttt{E}, \rho) = \texttt{C}_E \quad \text{stay}(\texttt{H}, \rho) = \texttt{C}_H \quad \texttt{E;H} \vdash \texttt{C} \rightsquigarrow \texttt{C}' \\ \texttt{E}';\texttt{H}' \models (\texttt{C}' \;\wedge\; \texttt{C}_E \;\wedge\; \texttt{C}_H)\end{array}}{\text{solve}(\texttt{E}, \texttt{H}, \texttt{C}, \rho) = \texttt{E}';\texttt{H}'} \qquad (\textsc{Solve})$$

$$\frac{\texttt{E = \{(x}_1\texttt{, v}_1\texttt{), ..., (x}_n\texttt{, v}_n\texttt{)\}} \quad \text{stay}(\texttt{x}_1\texttt{=v}_1, \rho) = \texttt{C}_1 \cdots \text{stay}(\texttt{x}_n\texttt{=v}_n, \rho) = \texttt{C}_n}{\text{stay}(\texttt{E}, \rho) = \texttt{C}_1 \;\wedge\; \cdots \;\wedge\; \texttt{C}_n} \qquad (\textsc{StayEnv})$$

$$\frac{\texttt{H = \{(r}_1\texttt{, o}_1\texttt{), ..., (r}_n\texttt{, o}_n\texttt{)\}} \quad \text{stay}(\texttt{r}_1\texttt{=o}_1, \rho) = \texttt{C}_1 \cdots \text{stay}(\texttt{r}_n\texttt{=o}_n, \rho) = \texttt{C}_n}{\text{stay}(\texttt{H}, \rho) = \texttt{C}_1 \;\wedge\; \cdots \;\wedge\; \texttt{C}_n} \qquad (\textsc{StayHeap})$$

$$\text{stay}(\texttt{x=c}, \rho) = \texttt{weak x=c} \qquad (\textsc{StayConst})$$

$$\text{stay}(\texttt{x=r}, \rho) = \rho\ \texttt{x=r} \qquad (\textsc{StayRef})$$

$$\frac{\texttt{x}_1 \text{ fresh} \cdots \texttt{x}_n \text{ fresh} \quad \text{stay}(\texttt{x}_1\texttt{=v}_1, \rho) = \texttt{C}_1 \cdots \text{stay}(\texttt{x}_n\texttt{=v}_n, \rho) = \texttt{C}_n}{\text{stay}(\texttt{r = \{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\texttt{\}}, \rho) = (\texttt{required H(r)=\{l}_1\texttt{:x}_1\texttt{,...,l}_n\texttt{:x}_n\texttt{\}}) \;\wedge\; \texttt{C}_1 \;\wedge\; \cdots \;\wedge\; \texttt{C}_n} \qquad (\textsc{StayObject})$$

Figure 1: Helper rules for solving constraints.

$$\frac{\begin{array}{c}\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{v} \\ \text{stayPrefix}(\texttt{E}, \texttt{H}, \texttt{L}) = \texttt{C}_L \quad \text{stayPrefix}(\texttt{E}, \texttt{H}, \texttt{I}) = \texttt{C}_I \\ \text{solve}(\texttt{E}, \texttt{H}, \texttt{C}_L \wedge \texttt{C}_I \wedge \texttt{L=v} \wedge \texttt{I}, \texttt{weak}) = \texttt{E}';\texttt{H}' \\ \texttt{E}';\texttt{H}' \vdash \texttt{C} \quad \text{solve}(\texttt{E}', \texttt{H}', \texttt{C} \wedge \texttt{L=v}, \texttt{required}) = \texttt{E}'';\texttt{H}''\end{array}}{\texttt{<E|H|C|I|L := e>} \longrightarrow \texttt{<E}''\texttt{|H}''\texttt{|C|I>}} \qquad (\text{S-}\textsc{Asgn})$$

The "stayPrefix" premises above impose a special form of stay constraints when solving identity constraints. These constraints are defined in Figure 2 and ensure that the l-value being assigned as well as the l-values in `I` will be updated deterministically. Specifically, if an l-value that may need to be updated has the form $\texttt{L}_0\texttt{.l}$, then the stayPrefix constraints ensure that the constraint solver will not change the value of $\texttt{L}_0$.

The next rule describes the semantics of object creation, which is straightforward. For simplicity we require a new object to be initially assigned to a fresh variable, so no constraint solving is required. This is no loss of expressiveness since that variable can immediately be used on the right-hand side of an arbitrary assignment.

$$\frac{\begin{array}{c}\texttt{o = \{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\texttt{\}} \\ \texttt{E;H} \vdash \texttt{e}_1 \Downarrow \texttt{v}_n \cdots \texttt{E;H} \vdash \texttt{e}_n \Downarrow \texttt{v}_n \\ \texttt{E(x)}\uparrow \quad \texttt{H(r)}\uparrow \quad \texttt{E}' = \texttt{E} \bigcup \{\texttt{(x, r)}\} \\ \texttt{H}' = \texttt{H} \bigcup \{\texttt{(r, \{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\texttt{\})}\}\end{array}}{\texttt{<E|H|C|I|x := new o>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C|I>}} \qquad (\text{S-}\textsc{AsgnNew})$$

The remaining rules are standard for imperative languages and are provided for completeness.

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

$$\frac{\begin{array}{c}\texttt{<E|H|C|I|s}_1\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>} \\ \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{|s}_2\texttt{>} \longrightarrow \texttt{<E}''\texttt{|H}''\texttt{|C}''\texttt{|I}''\texttt{>}\end{array}}{\texttt{<E|H|C|I|s}_1\texttt{;s}_2\texttt{>} \longrightarrow \texttt{<E}''\texttt{|H}''\texttt{|C}''\texttt{|I}''\texttt{>}} \qquad (\text{S-}\textsc{Seq})$$

$$\frac{\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{true} \quad \texttt{<E|H|C|I|s}_1\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}}{\texttt{<E|H|C|I|if e then s}_1\texttt{ else s}_2\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}} \qquad (\text{S-}\textsc{IfThen})$$

$$\frac{\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{false} \quad \texttt{<E|H|C|I|s}_2\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}}{\texttt{<E|H|C|I|if e then s}_1\texttt{ else s}_2\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}} \qquad (\text{S-}\textsc{IfElse})$$

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

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

3.3 Key Properties

We state two key theorems about the Babelsberg/UID core language. The proofs for these theorems can be found in the companion technical report [7]. The first theorem formalizes the idea that any solution to a value constraint preserves the structures of all objects and the relationships among objects:

Theorem. (Structure Preservation)
If $\texttt{<E|H|C|I|(once|always) C}_0\texttt{>} \longrightarrow \texttt{<E}'\texttt{|H}'\texttt{|C}'\texttt{|I}'\texttt{>}$ and $\texttt{E;H} \vdash \texttt{C}_0$ and $\texttt{E;H} \vdash \texttt{x : T}$,
then $\texttt{E}';\texttt{H}' \vdash \texttt{x : T}$.

The second theorem formalizes the idea that the result of an assignment statement is deterministic in terms of the resulting object structures:

$$\text{stayPrefix}(\texttt{E, H, x}) = \texttt{true} \qquad (\textsc{StayPrefixVar})$$

$$\frac{\texttt{L}=\texttt{x.l}_1\texttt{.}\ldots\texttt{.l}_n \quad n>0 \quad \texttt{E;H}\vdash\texttt{x}\Downarrow\texttt{r}_0 \quad \texttt{E;H}\vdash\texttt{r}_0\texttt{.l}_1\Downarrow\texttt{r}_1 \cdots \texttt{E;H}\vdash\texttt{r}_{n-2}\texttt{.l}_{n-1}\Downarrow\texttt{r}_{n-1}}{\text{stayPrefix}(\texttt{E, H, L}) = \texttt{x=r}_0 \wedge \texttt{r}_0\texttt{.l}_1\texttt{=r}_1 \wedge \cdots \wedge \texttt{r}_{n-2}\texttt{.l}_{n-1}\texttt{=r}_{n-1}} \qquad (\textsc{StayPrefixField})$$

$$\frac{\texttt{I} = \texttt{L}_1\texttt{==L}_2 \wedge \cdots \wedge \texttt{L}_{2n-1}\texttt{==L}_{2n} \quad \text{stayPrefix}(\texttt{E, H, L}_1) = \texttt{C}_1 \cdots \text{stayPrefix}(\texttt{E, H, L}_{2n}) = \texttt{C}_{2n}}{\text{stayPrefix}(\texttt{E, H, I}) = \texttt{C}_1 \wedge \cdots \wedge \texttt{C}_{2n}} \qquad (\textsc{StayPrefixIdent})$$

Figure 2: Additional stay constraints for solving identity constraints.

Theorem. (Structural Determinism)
If $\texttt{<E|H|C|I|L := e>} \longrightarrow \texttt{<E}_1\texttt{|H}_1\texttt{|C}_1\texttt{|I}_1\texttt{>}$ and
$\texttt{<E|H|C|I|L := e>} \longrightarrow \texttt{<E}_2\texttt{|H}_2\texttt{|C}_2\texttt{|I}_2\texttt{>}$ and
$\texttt{E;H}\vdash\texttt{x : T}_0$,
then there exists a type $\texttt{T}$ such that $\texttt{E}_1\texttt{;H}_1\vdash\texttt{x : T}$ and $\texttt{E}_2\texttt{;H}_2\vdash\texttt{x : T}$.

4. Evaluation

Our aim in developing a formal semantics for Babelsberg has been primarily a practical one, including clarifying the desired behavior of the language, 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 as discussed in the introduction, in response have adapted the existing Babelsberg/JS implementation to conform to it.

Beyond that, we offer two forms of evaluation. First, we have implemented our natural semantics in the Relational Meta-Language (RML) [20], yielding executable semantics both for the core Babelsberg/UID language presented above, and for Babelsberg/Objects [6], an extended, object-oriented language, which includes support for classes, methods, and constraint definitions that can include polymorphic method calls. We then evaluated a set of test programs in these executable semantics, using the Z3 solver [3] to provide heap and environment updates. As an extension to that, we provide a translation framework to generate test suites for the implementations in JavaScript, Ruby, and Smalltalk from our test programs. We have adapted the JavaScript implementation of Babelsberg and show that the practical implementation and semantics produce the same results for most of the tests, except those that test features of value classes, which are not available in JavaScript.

As a second form of evaluation, we have adapted the currently existing suite of example programs for Babelsberg/JS to the revised language. We demonstrate that nearly all of the programs continue to work without modification, or work with minimal adjustments. (These adjustments are described below.) Those programs that do not work anymore stopped doing so because they use constructs we explicitly decided to disallow.

4.1 An Executable Semantics

RML is a programming language to generate executables from natural language specifications. 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].

Z3 types 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 must 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. The last is encoded as a Z3 array that maps *labels* to reals. Labels and references are declared as finite domain types that range over the existing references and record labels respectively. We define the operation, comparison, and combination functions on that union type.

In reference [6], we provide 47 short test programs that illustrate various aspects of the semantics. We use these to validate that our semantics exhibits the characteristics we want. 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 older version of Z3 [23], but does not work in recent versions.) The remaining 44 all run and produce the expected results at each step of the execution.[5]

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 will 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, automatically translate them into the syntax of the target language using a simple mapping provided for each language, and wrap them

[5] The entire executable semantics and the test programs are available at `https://github.com/babelsberg/babelsberg-rml` and as an artifact with this paper.

in a test scaffold (also specific to each language) to execute them in each implementation.

All of the pre-existing implementations of Babelsberg require adaptation to pass these tests, because all deviate from our new semantics in one way or another. We began with the JavaScript implementation, as it is the most advanced, and updated it based on the work described here. After modification, of the 44 test cases that work in the RML semantics, 41 produce the expected results. The remaining three specifically test properties of value classes in constraints. Since JavaScript does not support value classes, and since we do not currently 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, due to insufficient support for explicit identity constraints and use of an older version of Z3 that does not include soft constraints.

4.2 Applying the Semantics to Babelsberg/JS

Adapting Babelsberg/JS to follow the semantics presented here required changes to both the translation from imperative expressions to constraints (called *constraint construction* in Babelsberg/JS) and the way in which the solvers are called, as well as changes to some of the existing example programs.

Modifications to the Implementation

First, we modified the code that translates JavaScript expressions into constraints to only inline methods that consist of a simple return expression. Methods that have any other statements in them are executed and their result returned, rather than being inlined, so they can only be used in the forward direction. We did not modify the translation to prohibit creating new objects — as described in Section 2.4.2, 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. As an optimization, Babelsberg/JS omits re-translating expressions if no dependent variables changed. We have left this optimization in place.

The second change added 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 this, and in its place add a restricted form of identity constraints. 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 [9] to propagate identity changes as they occur, and call any solvers for value constraints after that. Babelsberg/JS, through its implementation of a cooperating solvers architecture, already supports solving in multiple phases. 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. 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.

Modifications to Existing Programs

One of our strategies in this work has been to start with clear and simple rules that are understandable to the programmer and amenable for use in proofs, recognizing that some of these rules would likely be too conservative. We next assess where the key limitations are by testing them with a set of typical programs[6], finding which ones no longer work, and classifying the problems that arise frequently. We now summarize the problems that we discovered, and also discuss possible directions for amending our rules to be more convenient. However, in considering such amendments, we need to balance simplicity and clarity with the needs for additional expressiveness. In particular, we do not want to introduce a large number of additional special cases or forms, and in any case we need to retain soundness.

The programs we examine were constructed using the LivelyKernel's direct manipulation Morphic interface, and include a graphical application for building and simulating electrical circuits, games, a GUI for a color picker, and tools for temperature conversion. They are about 200 lines of code each, with between ¼ and ½ of those being constraint definitions. These programs illustrate the division of labor and interplay between the imperative object-oriented portions of the code and the constraint parts, as well as the different kinds of constraints and solvers that can be employed. The electrical circuit editor and simulator [5], for example, uses constraints and an appropriate solver to capture the relevant laws of physics such as Ohm's Law and Kirchhoff's Current Laws and to solve the resulting sets of simultaneous linear equations, and imperative constructs to support actions such as making new resistor, battery, and meter instances, and placing and connecting them. The color picker and temperature converter [5] illustrates hard and soft constraints, as well as support for fast incremental re-solving during interactive editing.

We present each issue, its intended semantics, a workaround with our current semantics, and directions for future work that could address the issue.

[6] The examples we used are part of a published artifact [5], and are also available from the Babelsberg/JS repository at `https://github.com/babelsberg/babelsberg-js`. We used the versions from the repository.

Argument Checking 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 are examples of the surprising behaviors we want to avoid. As an example, consider the frequently used method to add two points in LivelyKernel:

Listing 6.

```
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);
}
```

We almost certainly don't want to satisfy a constraint that calls the `addPt` method with a bad argument by changing the argument to be a point. Instead, the argument check should not be part of the constraint.

As a workaround, although it is not semantically clean, the practical implementation does allows such tests on arguments, because these checks are so pervasive that removing them was very inconvenient. As part of future work, we may want to support assertions as constructs that can be checked before handing the constraints to the solver. Assertions can then be used instead of branches and explicit exceptions.

Branching We encountered a similar issue with methods that return one of two expressions, depending on a test. Our semantics does not allow branching in multi-way constraints. Such a method encountered frequently is `getPosition`:

Listing 7.

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

The desired semantics is that we can use `getPosition` in a constraint, either to access the current position or to modify the object's position, but not to modify the object's `FixedPosition` flag. However, the builtin `getPosition` was not written anticipating its possible use in constraints, and so `FixedPosition` is not annotated as read-only. As an earlier workaround, the programmer was able to select a particular solver to get the desired behavior. Cassowary, for example, cannot reason about booleans and treated the flag as a constant, as intended. Z3, on the other hand, would sometimes change `FixedPosition`, leading to surprising solutions and graphical glitches.

In our semantics, the specifics of the solver no longer influence the meaning of using `getPosition` in a constraint — it can only be used in the forward direction. As a workaround for using it multi-directionally, the developer can move the test outside of the method and add the constraint only for the branch that is chosen. In future work, we want to add a mechanism to recognize this pattern and automatically mark the branch condition as read-only to the solver.

Benign Side Effects Lazy initialization and caching are used in some of the example applications. We explicitly allowed such benign side effects in prior work [4, 5], but our formal semantics now disallows them. (We believe that this is an appropriate simplification for the formal semantics, preferring to leave this issue to clear but less formal guidance to programmers and language implementors.) For example, the LivelyKernel method `Morph.getBounds` is used 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:

Listing 8.

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

The desired semantics is to bypass the cache and affect the actual fields that contribute to the calculation of the bounds.

A workaround here is to call `innerBounds` directly, and circumvent the caching. In the practical applications, this change had the additional benefit of removing some UI glitches that happened when the solver changed the cached bounds, but the rendering code was watching the inner bounds. As future work, we want to introduce rules in the semantics to recognize cached values as benign side-effects, so the solver modifies the correct fields and invalidates or updates the cache appropriately.

Local Variables Some methods use local variables to split up calculations or name constants. We explicitly allowed such methods in the original Babelsberg design, as long as the variables were used in static-single-assignment (SSA) fashion [4].

Listing 9.

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

Although the desired semantics in this case is to inline the variables as constants into the return expression, our experience with OCP has shown that it is not always clear if the system should change the value of hidden variables if they are used to build a constraint system, or if they can always be regarded as constants. We thus argue that, although sometimes inconvenient, the restriction to disallow local variables makes it easier to reason about the programs.

As a workaround, manual inlining of the split up calculation into the return expression was required. As an extension to our rules, when local variables are used only as names for constants, these should be inlined automatically.

Overall, we think that 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.

5. Application to Related Work

There are a large number of related systems that integrate constraints with imperative languages, ranging from constraint satisfaction libraries and domain specific languages (DSLs) to Object-Constraint Programming languages that offer syntactic and semantic integration. How these systems choose to balance the power of solvers and how aware the programmers need to be of possible non-determinism varies, as does the extent to which programmers are able to express constraints involving arbitrary mutable objects, object identity, or methods.

The work presented here allows object-constraint languages to use different constraint solver libraries as black box solvers and ensures that using a different solver does not introduce non-deterministic and surprising behaviors regarding the structure and identity of objects. A large number of constraint satisfaction libraries exist[7], however, these systems do not integrate syntactically or semantically with the host language. They require explicit action by the developer to copy and convert variables between the imperative and declarative systems, require the developer to explicitly ensure that the solver is called at appropriate times, and require explicit action to deal with updates to the object graph. Thus, our work is not directly relevant to those systems by themselves, but allows them to be used as solvers in an OCP language.

Kaleidoscope [8, 16] was an early Constraint Imperative Programming (CIP) language, and had to address issues that arise from integrating declarative constraints and imperative code with mutable state. Kaleidoscope uses soft constraints to ensure that, in the absence of other constraints, the solver does not change the values of existing variables, which we also encode in our semantics. In contrast to the work presented here, Kaleidoscope separates methods that generate constraints from ordinary methods. This avoids having to give rules for how a method can be used in constraints, but it puts the burden on the programmer to develop and maintain two sets of interfaces, one for use in constraints and one for use in imperative statements.

Kaleidoscope'93 also included support for identity constraints [16], to express the distinction in object-oriented languages between object equality and identity. However, Kaleidoscope'93 allowed the solver to determine the identity of method arguments when solving value constraints, and used multi-method dispatch to look up methods in a constraint. In combination this gave the system much power, but also made it hard to understand which solution the system might select. In our semantics constraints do not use multi-method dispatch, removing one dimension of freedom from the solver (as well as conforming to a more standard semantics for an object-oriented language) and in addition the solver is not allowed to change the identity or structure of arguments for value constraints.

Turtle [11] is a more recent CIP language written from scratch, while Kaplan [14] provides constraints in Scala. Both separate the declaration of *constrainable* variables from ordinary variables to make it clearer what may happen when a variable is used. Neither includes support for identity constraints, however. Like Babelsberg, the Turtle system provides constraint priorities; Kaplan does not. Because ordinary variables in Turtle are not determined by the solver, only constrainable variables have low-priority stay constraints on them. Kaplan does not currently support constraints over mutable types, so stay constraints are not relevant for it. Analogous to Kaleidoscope and in contrast to OCP, both languages separate constraint functions from ordinary functions. In Kaplan, only such specifically annotated functions can be used in constraints. Turtle does allow ordinary methods and variables in constraints; however, their values are treated as constants, making all ordinary methods work only in the forward direction. We believe the simple rules for method inlining in our semantics could be used with these systems as well to allow a restricted set of ordinary object-oriented methods to be used in constraints.

BackTalk [19, 21] and SQUANDER [17] explicitly allow the constraint solver to determine object structure, and even to create new objects that satisfy a set of constraints. While these systems have many differences, both use object-oriented methods in constraints as tests for a backtracking algorithm that tries different objects and object structures as assignments for the constrained variables. Thus, it is a basic property of these systems to allow actions that we prohibit in OCP. In those systems, constraints are most useful when the concrete identity of an object is not relevant. When declarative code interacts with imperative code, however, this may not be the case, especially in dynamic languages such as JavaScript, where libraries add and remove fields to specific

[7] See, for example, this catalog: `http://openjvm.jvmhost.net/CPSolvers/`, accessed March 9, 2015

instances at runtime and check for object identity rather than fields. Our structural type-checks could be used to limit the amount of non-determinism in these systems by ensuring that all solutions to a constraint must have the same structure.

Mozart/Oz [22] is a multi-paradigm language that supports functional, imperative, and logic programming. It also has a special focus on supporting concurrency, non-determinism, and search. In Oz, constraint programming is much more explicit than in an object-constraint language, with explicit variable types for unbound and finite-domain variables. Oz uses a constraint store that is monotonic, meaning that bindings and constraints can be added but not removed or changed, which is something we explicitly support in the work presented here. Another difference is that Oz explicitly allows non-determinism in constraint solving. Two features of Oz that we do not have are explicit integration of threads of concurrent execution and constraint solving, and support for backtracking (although we have a design for the latter, which we will be investigating as part of future work).

Finally, αRby [18] is a language that embeds the Alloy specification language [12] in Ruby. Its goal is to allow Alloy users to easily pre- or post-process their models using imperative libraries, for example to experiment with visualizations for Alloy models. αRby translates Ruby programs into Alloy, but the programs are written in a DSL that closely mimics the Alloy language rather than using ordinary Ruby classes and methods. In contrast to OCP, αRby aims to provide imperative constructs to Alloy users, that is, programmers familiar with constraints. The restrictions on the solver we introduce in this work for the benefit of imperative programmers are thus not desirable in αRby.

6. Conclusion and Future Work

We have presented design principles to control the power of the solver in object-constraint programming languages to avoid surprising or non-deterministic behavior by ensuring that object structure and identity are preserved when adding constraints, and that any changes to them are deterministic. We also presented a small set of syntactic rules that guide developers in expressing constraints. Our principles are formalized in a natural semantics for an object-constraint language that preserves key properties of existing object-constraint implementations. For this semantics we presented two theorems that formalize two key properties of the first principle, namely that we never allow the solver to find solutions that would needlessly add or remove fields from existing objects. Our rules also ensure that the only way for a variable to change its structure is through an assignment statement — thus, solving constraints outside of an assignment can never lead to solutions in which variables change their structure.

The semantics given here defines a useful subset of the language. A full object-constraint language that supports standard classes and methods is described in reference [6]. This description includes formal rules for defining constraints on the results of message sends that implement our informal description in Section 2.4.3. Its restrictions are easy to understand, but may be too conservative, as shown by our experiences with existing Babelsberg example programs. One direction for future work will be to allow somewhat more general kinds of expressions in methods called from constraints to be used multi-directionally.

We also presented an executable semantics that can run a suite of example programs, provided a mechanism to generate a language test suite to check the implementations' conformance to the semantics, and so far modified one existing implementation to follow our design, thus demonstrating its practicality. Another direction for future work is to use this test suite to move other implementations closer to conforming to the semantics, to further test its practicality.

Acknowledgments

Many thanks to our colleagues in the Communications Design Group for much useful feedback on this work. This research was funded by the Hasso Plattner Institute, SAP, and Viewpoints Research Institute.

References

[1] N. Björner and A.-D. Phan. νZ–maximal satisfaction with Z3. In *Symbolic Computation in Software Science*, 2014.

[2] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *LISP and Symbolic Computation*, 5(3):223–270, 1992.

[3] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer, 2008.

[4] T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and solving constraints on object behavior. *Journal of Object Technology*, 13(4):1–38, 2014.

[5] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/JS: A browser-based implementation of an object constraint language. In *ECOOP*, pages 411–436. Springer, 2014.

[6] T. Felgentreff, T. Millstein, and A. Borning. Developing a formal semantics for Babelsberg: A step-by-step approach. Technical Report 2014-002b, Viewpoints Research Institute, 2015. Available at `http://www.vpri.org/pdf/tr2014002_babelsberg.pdf`.

[7] T. Felgentreff, T. Millstein, A. Borning, and R. Hirschfeld. Checks and balances — constraint solving without surprises in object-constraint programming languages: Full formal development. Technical Report 2015-001, Viewpoints Research Institute, 2015.

[8] B. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In *ECOOP*, pages 268–286, June 1992.

[9] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[10] M. Graber, T. Felgentreff, R. Hirschfeld, and A. Borning. Solving interactive logic puzzles with object-constraints — an experience report using Babelsberg/S for Squeak/Smalltalk. In *Workshop on Reactive and Event-based Languages & Systems*, 2014.

[11] M. Grabmüller and P. Hofstedt. Turtle: A constraint imperative programming language. In *RDIS*, pages 185–198. Springer, 2004.

[12] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[13] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM, 1987.

[14] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL*, pages 151–164. ACM, 2012.

[15] G. Lopez, B. Freeman-Benson, and A. Borning. Constraints and object identity. In *ECOOP*, pages 260–279. Springer, 1994.

[16] G. Lopez, B. Freeman-Benson, and A. Borning. Kaleidoscope: A constraint imperative programming language. In *Constraint Programming*, volume 131 of *NATO ASI Series, Series F: Computer and System Sciences*, pages 313–329. Springer, 1994.

[17] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520. ACM/IEEE, 2011.

[18] A. Milicevic, I. Efrati, and D. Jackson. αRby–an embedding of Alloy in Ruby. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2014.

[19] F. Pachet and P. Roy. Integrating constraint satisfaction techniques with complex object structures. In *Conference of the British Computer Society Specialist Group on Expert Systems*, pages 11–22. Cambridge University Press, 1995.

[20] M. Pettersson. RML—a new language and implementation for natural semantics. In *PLILP*, pages 117–131. Springer, 1994.

[21] P. Roy, A. Liret, and F. Pachet. A framework for object-oriented constraint satisfaction problems. In *Computing Surveys Symposium on Object-Oriented Application Frameworks*, pages 1–22. ACM, 2000.

[22] P. Van Roy, P. Brand, D. Duchier, S. Haridi, C. Schulte, and M. Henz. Logic programming in the context of multiparadigm programming: The Oz experience. *Theory and Practice of Logic Programming*, 3(06):717–763, 2003.

[23] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE*, pages 114–124. ACM, 2013.