Developing a Formal Semantics for Babelsberg: A Step-by-Step Approach (DRAFT)

Tim Felgentreff, Todd Millstein, and Alan Borning

VPRI Technical Report TR-2014-002

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

Developing a Formal Semantics for Babelsberg: A Step-by-Step Approach

Tim Felgentreff, Todd Millstein, and Alan Borning

VPRI Technical Report TR-2014-002
Draft of July 13, 2014

1 Introduction

Babelsberg [4] is a family of object constraint languages, with current instances being Babelsberg/R (a Ruby extension), and Babelsberg/JS [5] (a Javascript extension). The Babelsberg design has evolved alongside with its implementations. However, because the implementations were driven in part by practical considerations about the expectations of programmers familiar with the underlying host language, this has led to a number of semantic choices that are muddled with implementation specifics. For example, there have been a number of confusing issues with respect to constraints and object identity, how to represent assignment in the solver, and the effects of the solver not finding a solution to a set of constraints. In an effort to understand these better and provide a complete design that instances of Babelsberg can implement, we give here a formal operational semantics for Babelsberg.

We've found it helpful to approach the problem incrementally, first devising a formal semantics for a very simple constraint imperative language, and then building up to a full object constraint language. In this memo we present the semantics in that fashion as well. The languages are as follows:

- Babelsberg/PrimitiveTypes (the only datatypes are booleans plus a set of primitive types, such as integers, reals, and strings). The concrete typed variant we present is Babelsberg/Reals.
- Babelsberg/Records (Babelsberg with immutable records, along with primitive types).
- Babelsberg/UID (Babelsberg with mutable records that live on the heap and so have an identity, can be aliased, etc., as well as primitive types).
- Babelsberg/Objects (Babelsberg with mutable objects, classes, methods, messages, and inheritance)

In each case, we first provide an informal discussion and examples, and then the formal semantics. This tech report is intended to accompany a paper to be submitted for conference publication, which will focus just on Babelsberg/Objects, perhaps with a brief introductory example using Babelsberg/Reals.

2 Motivation

Our formal semantics are intended to provide a complete semantics of Babelsberg that can be used to inform a practical implementation of the language. It is meant to as simple as possible, while still encompassing the

major design decisions needed to guide language implementers. Because Babelsberg is a design to provide object-constraint programming in an object-oriented host language, the semantics omits some constructs such as exception handling for constraint solver failures and syntactic sugar that are intended to be inherited from the host language. The semantics instead focus on the expression of standard object-oriented constructs that need to be modified to support the Babelsberg design.

An overarching design goal is that in the absence of constraints, Babelsberg should be a standard object-oriented language. We have thus resisted the temptation to add other interesting features, and to make the smallest possible changes while still accommodating constraints in a clean and powerful way.

A heuristic for the design is that when there is a choice, we favor simplicity over power (at least power to do interesting operations in the language that have nonetheless not yet proven themselves useful in practice). The constraint imperative language Kaleidoscope [6, 7, 8, 9], particularly the early versions, was arguably too complex in part because it was too powerful in interesting ways of just this kind, making it difficult to understand what the result of a program might be and also difficult to implement efficiently. This also makes the design more independent of the particular host language and solvers used in implementing it, because only a small set of basic operations have to be adapted.

3 Constraints

The semantics for each of these languages includes a step in which we assert that some set of values for variables is a correct solution to a collection of hard and soft constraints. There is a constraint solver that is a black box as far as the rest of the formal semantics is concerned, and that handles all the issues regarding finding a solution, dealing with conflicting soft constraints, and so forth. The solver should be sound but may be incomplete (i.e., it should never return an incorrect answer, but might respond that the set of constraints is too difficult for it to determine whether or not there is a solution).

We use the semantics for hard and soft constraints presented in [3]. An earlier paper on Babelsberg [4] has a description of the relevant theory as well.

The solver should find a single best solution — if there are multiple solutions, the solver 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 [11], is left for future work — see Appendix A.3.)

The way we trade off conflicting soft constraints is defined by a *comparator*. The two most relevant comparators are locally-predicate-better (LPB) and weighted-sum-better (WSB). Locally-predicate-better only cares whether a constraint is satisfied or not, not how far off the value is from the desired one. Any Pareto-optimal solution is acceptable. For example, a solution that satisfies one weak constraint A and violates three weak constraints B, C, and D is OK, as long as there isn't a solution that satisfies both A and some additional constraint, even if there is another solution that satisfies B, C, and D but not A. The DeltaBlue solver [10] finds LPB solutions. Weighted-sum-better considers the weighted sum of the errors of constraints with a given soft priority, and picks a solution that minimizes the sum. If there is more than one solution and there are additional lower priority constraints, we then consider the lower-priority ones to winnow down the possible solutions, priority by priority. For this comparator we need an error in satisfying a constraint, which should be 0 iff the constraint is satisfied. Cassowary [1] finds WSB solutions.

Here are two examples. (These are described from the point of view of the declarative theory of hard and soft constraints, not with respect to how an actual solver can find that solution.)

required	$x + y = 10$
strong	$x = 8$
weak	$y = 0$

The required constraint has an infinite number of solutions. When we winnow these down to solutions that satisfy the strong constraint, there is only one left: $x = 8$, $y = 2$. This is both a LPB and a WSB solution. The weak constraint has no impact on the solution in this case.

Now consider:

required	$x + y = 10$
strong	$x \geq 5$
weak	$y = 20$

We'll only consider the WSB comparator this time, since it is more suitable for use with inequalities. (DeltaBlue does not handle inequalities.) Again, the required constraint has an infinite number of solutions. We winnow these down with the strong constraint to all $x \in [5, \infty)$, y such that $x + y = 10$. The weak constraint is unsatisfiable, so we minimize its error, resulting in the solution $x = 5$, $y = 5$.

Strict inequality constraints with metric comparators can be problematic in the presence of soft constraints, since they can lead to no solutions. Consider:

required	$x > 10$
weak	$x = 5$

This set of constraints has no solution — for any potential solution that satisfies $x > 10$, we can find another that better satisfies $x = 5$. For this reason, our examples usually use non-strict inequalities.

The "required" priority is special, in that those constraints must be satisfied in any solution. Both the semantics of hard and soft constraints, and the Cassowary and DeltaBlue solvers, can handle arbitrary numbers of soft priorities. However, for simplicity in the remainder of this note, we only use two, namely "strong" and "weak".

We can also annotate variables used in constraints as *read-only*. Intuitively, when choosing the best solutions to a set of constraints with priorities, constraints should not be allowed to affect the choice of values for their read-only variables, i.e., information can flow out of the read-only variables, but not into them. There is a formal, declarative definition of read-only annotations in [3], which was in turn adapted from that in the ALPS flat committed-choice logic language [12]. A *one-way constraint* can be represented by annotating all but one of the constrained variables as read-only. One can also annotate an expression as read-only — this is syntactic sugar for annotating each of its variables as read-only.

In all of the languages described in the sections that follow, the constraint solver satisfies the constraints at each time step, but does not itself need to know about time. Instead, time is built into the semantic rules; as far as the solver is concerned, time is handled at each step by adding weak *stay constraints* that variables keep their old values. In other words, the semantic rules handles time, while the solver handles solving the constraints including both hard and soft constraints.

3.1 Conjunctions and Disjunctions of Constraints

In general, a constraint might consist of conjunctions, disjunctions, and negations of atomic constraints. For a conjunction or disjunction, if there is a priority, it applies to the entire constraint, not to components. Thus this is legal:

strong $(x = 3 \vee x = 4)$

but this is not allowed:

(strong $x = 3$) $\vee$ (weak $x = 4$)

Only some solvers, such as Z3, can accommodate disjunctions and explicit conjunctions of constraints. For DeltaBlue and Cassowary, conjunctions of constraints are implicitly specified by feeding multiple constraints to the solver, while disjunctions aren't allowed. However, as noted above, the solver is a black box as far as the rest of the formal semantics is concerned.

3.2 Using Multiple Comparators

The semantics for hard and soft constraints used in work to date has always employed a single comparator to specify how to trade off conflicting soft constraints. However, we can extend the theory to allow multiple comparators — different priorities can use different comparators. For example, we might use LPB for comparing the solutions to strong constraints, and WSB for the weak ones. We will use this extension later so that we can mix constraints on object identity and object values. (Identity constraints will be first introduced for Babelsberg/UID (Section 6), and also used for Babelsberg/Objects (Section 7.) An identity constraint is either satisfied or it's not — a WSB comparator isn't appropriate for them. But there may also be constraints on values, for example on the x and y fields of points, for which we do want to use a WSB comparator. As long as we give different priorities for the identity and the value constraints, this extension allows doing this.

4 Babelsberg/Reals and Babelsberg/PrimitiveTypes

We start with a very basic language, Babelsberg/Reals, that has only primitive values. In Babelsberg, constraints are expressions that return a boolean – the constraint solver's task is to get the expression to evaluate to true. So Boolean is a required datatype for all Babelsberg languages. In addition, we add reals as a second primitive type. Here are some examples.

```
x := 3;
x := 4;
always x>=10
```

Babelsberg/Reals, as with the other languages, uses two different execution modes: *imperative execution mode* and *constraint construction mode*. For ordinary statements, expressions are evaluated in imperative execution mode. This is mostly standard, except that assignment is syntactic sugar for a `once` constraint (i.e., a constraint that is asserted and then retracted). This `once` constraint is formed by evaluating the expression on the right hand side of the assignment, and then constraining the variable on the left hand side of the assignment to be equal to the result. In contrast, constraint construction mode is used with constraints that are created by an explicit `always` or `once` constraint. The semantic model includes a *constraint store* consisting of closures corresponding to constraints encountered in an `always` or `once` statement: when an `always` or `once` statement is encountered, the interpeter (as specified by the formal semantics) constructs a closure over the expression's environment of definition and saves it in the constraint store. For `always`, the closure remains for the duration of the program's execution; for `once`, it is removed after the constraints are satisfied.

After evaluating the first statement we hand the following `once` constraint to the solver to find a value for `x`:

required $x = 3$

The solver finds a value for x, which is then used to update the environment to be `x` $\mapsto$ `3`.

Note that programs are written in fixed pitch font, while solver constraints are in math font, with the priority in normal font. This reflects that the statements in the program are different semantically from the constraints in the language provided to the solver. For Babelsberg/Reals the translation is trivial, but will become non-trivial in subsequent languages. Finally, the variable names and values in the environment are again in fixed pitch font, since these are variable names from the program and constants in the Babelsberg/Reals language.

Continuing with the example, after the second statement we hand the following constraints to the solver:

weak	$x = 3$
required	$x = 4$

The weak $x = 3$ constraint is the stay constraint that `x` retain its previous value, while the required $x = 4$ constraint comes from the second assignment. This has the solution $x = 4$, resulting in a new environment `x` $\mapsto$ `4`.

The third statement adds a closure representing the `always` constraint to the constraint store. For Babelsberg/Reals, the translation from a constraint in the source language to one in the solver language is trivial. (However, it will be more complex in later languages.) So after that statement we have the following constraints:

weak	$x = 4$
required	$x \geq 10$

If we use a metric comparator such as weighted-sum-better (WSB) we get the solution $x = 10$, since this minimizes the error for the weak constraint. If we use LPB, then any $x \in [10, \infty)$ is a solution, and the system is free to select any of them. (However, as noted above, typically we wouldn't use LPB if we have inequalities.)

Here is another example that involves using a variable on the right hand sides of assignment statements, as well as showing the interaction of assignments with `always` constraints.

```
x := 3;
always y = x+100;
x := x+2
```

After evaluating the first statement and solving the resulting constraint, the environment has the binding `x` $\mapsto$ `3`. The second statement causes the constraint `always y = x+100` to be added to the constraint store. We then hand the following constraints to the solver to find values for `x` and `y`:

weak	$x = 3$
required	$y = x + 100$

As before, the weak $x = 3$ constraint is the stay constraint that `x` retain its previous value. This has the solution `x` $\mapsto$ 3, `y` $\mapsto$ 103.

After evaluating the third statement, we have the following constraints:

weak	$x = 3$
required	$y = x + 100$
required	$x = 5$

The first constraint is the weak stay on `x`, the second comes from the always constraint in the constraint store, and the third comes from the assignment `x:=x+2` (where we evaluated `x+2` in the old environment to get 5). After solving these constraints, we have `x` $\mapsto$ 5, `y` $\mapsto$ 105.

An interesting aspect of this semantics, both as presented informally above and in the formalism that follows, is that we no longer model assignment as a constraint on variables at different times. (In the refinment model, as presented in the first Babelsberg paper [4], we described an assignment `x := x+2` as a constraint $x_{t+1} = x_t? + 2$.) But now the constraint solver doesn't know anything about time. Instead, we simply solve constraints in the current environment — the effect of considering the value of `x` at different times is achieved by first evaluating the right hand side of an assignment in the old environment, and then asserting `once` constraints that cause variable values to be updated as needed.[1]

The program might also introduce simultaneous equations and inequalities. For example:

```
x := 0;
y := 0;
z := 0;
always x+y+2*z = 10;
always 2*x+y+z = 20;
x := 100
```

Assuming the solver can solve simultaneous linear equations, after the final assignment we will have `x` $\mapsto$ `100`, `y` $\mapsto$ `-270`, `z` $\mapsto$ `90`.

As an example of unsatisfiable constraints, consider:

```
x := 5;
always x<=10;
x := x+15
```

After evaluating the first statement the environment includes the binding `x` $\mapsto$ `5`. After evaluating the statement that generates the `always` constraint, we solve the constraints

weak	$x = 5$
required	$x \leq 10$

This has the solution $x = 5$. Then we evaluate the last assignment, resulting the constraints

weak	$x = 5$
required	$x \leq 10$
required	$x = 20$

Note that the required $x \leq 10$ constraint has persisted into this new set of constraints. These constraints are unsatisfiable.

This behavior is the same as that of the refinement model, in which the program would be equivalent to these constraints (which are also unsatisfiable):

$$\begin{array}{rrcl} & x_0 & = & 5 \\ \forall t > 0 & x_t & \leq & 10 \\ \text{weak} & x_1 & = & x_0? \\ \text{weak} & x_2 & = & x_1? \\ & x_2 & = & x_1? + 15 \end{array}$$

[1] These models are equivalent, at least for Babelsberg/Reals, so this is just a question of presentation. In the refinement model, when we represent an assignment as a constraint among variables at time t on the right hand side, and a variable at $t + 1$ on the left hand side, we annotate all the variables on the right hand side as read-only, which yields the same result as first evaluating the right hand side in the old environment.

However, it is *different* from the behavior of the perturbation model, in which we would assign 20 to x on the last statement, and then bump it down to 10 to satisfy the `always` constraint, rather than moving to the fail state.

4.1 Requirements for Constraint Expressions

The expressions that define constraints have a number of restrictions. These will apply to all of the Babelsberg languages.

1. Evaluating the expression that defines the constraint should return a boolean. (This is checked dynamically.)
2. The constraint expression should either be free of side effects, or if there are side effects, they should be benign, for example, doing caching. (This isn't checked.)
3. The result of evaluating the block should be deterministic. For example, an expression whose value depended on which of two processes happened to complete first wouldn't qualify. (This does not arise in the toy languages here, although we do need this restriction for a practical one.)

4.2 Control Structures

Babelsberg/Reals includes `if` and `while` control structures. These work in the usual way, and allow (for example) a variable to be incremented only if a test is satisfied, or an `always` constraint to be conditionally asserted. The test for an `if` statement is evaluated, and one or the other branch is taken — there is no notion of backtracking to try the other branch. (Adding Prolog-style backtracking is left for future work, and would be supported for goals rather than imperative control structures — see Appendix A.3.) Similarly, a `while` statement executes the body a fixed number of times — there is no possibility of backtracking to execute it a different number of times.

The test in an `if` or `while` statement can use short-circuit evaluation when evaluating an expression involving `and` and `or`. For example, this program results in `x` $\mapsto$ 100 (and doesn't get a divide-by-zero error):

```
x := 4;
if x=4 or x/0=10
  then x := 100
  else x := 200
```

On the other hand, conjunctions or disjunctions in constraints that are evaluated in constraint construction mode, and that then generate boolean constraints for the solver, are not evaluated using short-circuit evaluation — everything gets encoded in the expression in the solver language, and then turned over to the solver. Also, separate from `if` statements, we could have `if` expressions in constraints. (We don't actually have `if` expressions in Babelsberg/PrimitiveTypes, since there is a simple translation into conjunctions and disjunctions, but it's useful to note that these are nevertheless different concepts.)

For example, if we had `if` expressions in constraints, this constraint would have the solution `x` $\mapsto$ 10:

```
always if x=4 then x=5 else x=10
```

It is equivalent to this constraint:

```
always (x=4 and x=5) or (x!=4 and x=10)
```

Here is an example of an unsatisfiable constraint of this sort:

```
always if x=4 then x=5 else x=4
```

which is equivalent to:

```
always (x=4 and x=5) or (x!=4 and x=4)
```

4.3 Adding Other Primitive Types

It is straightforward to extend Babelsberg/Reals with other primitive types, such as integers and strings. When we need to refer to this language rather than just Babelsberg/Reals we will call it Babelsberg/PrimitiveTypes. Note that all the types in Babelsberg/PrimitiveTypes are atomic — we don't have recursive types or types that define values that hold other types (such as records, arrays, or sets).

4.4 Formalism

We present the formal semantics of Babelsberg/PrimitiveTypes. Besides the program, there is an *environment* that maps names to values.

4.4.1 Syntax

Statement	`s`	::=	`skip` | `x := e` | `always C` | `once C` | `s;s` | `fail` | `if e then s else s` | `while e do s`
Constraint	`C`	::=	`e` | ρ `e`
Expression	`e`	::=	`c` | `x` | `x?` | `e` $\oplus$ `e` | `e` $\oslash$ `e` | `e` $\otimes$ `e`
Constant	`c`	::=	`true` | `false` | `nil` | base type constants
Variable	`x`	::=	variable names
Value	`v`	::=	`c`

The language includes a set of boolean and real constants and the undefined value `nil`, ranged over by metavariable `c`. A finite set of operators on expressions is ranged over by $\oplus$. For booleans this set includes no operations, but it does include operations on the reals such as `+`, `*`, and `✓`. The symbol $\oslash$ ranges over a set of *predicate* operators ($=$ and $\neq$ for booleans, $\leq$, $<$, $=$, and so on for reals.) These test properties of base types in the language. The symbol $\otimes$ ranges over a set of logical operators for combining boolean expressions (e.g., $\wedge$, $\vee$). The predicate operators are assumed to include at least an equality operator `=` for each primitive type in the language, and the logical operators are assumed to include at least conjunction $\wedge$. The syntax of this language does have some limitations as compared with that of a practical language — for example, there are only binary operators (not unary or ternary), and the result must have the same type as the arguments. We make these simplifications since the purpose of Babelsberg/PrimitiveTypes is to elucidate the semantics of such languages as as a step toward Babelsberg/Objects, rather than to specify a real language.

For constraints, the symbol ρ ranges over a finite and totally ordered set of constraint *priorities* and is assumed to include a bottom element *weak* and top element *required*. A constraint with no explicit priority is assumed to have the priority *required*.

The syntax is thus that of a simple, standard imperative language except for the `always` and `once` statements, which declare constraints, and the `?` annotation for variables. An `always` constraint must hold for the rest of the programs execution, whereas a `once` constraint is satisfied by the solver and then retracted. The `?` indicates that a variable is read-only if used in a constraint, i.e., the programmer does not allow the solver to change the variables in the expression to satisfy constraints. Finally, the `fail` statement is not intended for use at the source level, but rather is used in the semantics to make explicit that the program halts due either to an unsatisfiable constraint or to the solver being unable to determine whether the constraint is satisfiable. (The formal semantics doesn't distinguish between these two cases, but lumps them both together as `fail`. In a practical implementation, we would likely want to distinguish them, since it's useful if we can inform the programmer that the constraints are truly not satisfiable. We could also add standard exception handling to remove the unsatisfiable or unknown constraint and continue, but omit this here for simplicity.)

4.5 Syntax of the Solver Language

The Babelsberg language is parameterized by a particular constraint solver that is used to find a solution to the constraints. In our formalism we use the following syntax as the language of the solver:

4.6 Syntax of the Solver Language

Constraint	C	::=	$e \mid \rho\ e$
Expression	e	::=	$c \mid x \mid x? \mid e \oplus e \mid e \oslash e \mid e \otimes e$
Constant	c	::=	numeric and boolean constants
Variable	x	::=	variable names
Value	v	::=	c

The elements of the solver's language are analogous to the syntax of allowed constraints in the source language for Babelsberg/PrimitiveTypes defined above. Note, however, that the elements of the solver language are really distinct entities from those in the source language. We will see that they have a straightforward mapping, but they are not the same values that happen to be written in a different font [2].

Some solvers may only support a subset of the above syntax, for example requiring constraints to be in some canonical form, or lacking support for disjunctions of constraints, prioritized constraints, or some particular primitive type (e.g., strings). On the other hand, some solvers may support constraints that are not in our syntax, such as constraints on records or objects. For generality we assume that all constraints must be translated to the simple language above. As we progress to languages with additional constructs, such as records and then objects, we will however retain the above solver language.

4.6.1 Semantics

The semantics is defined by several judgments, defined below. These judgments depend on the notion of an *environment*, which is a partial function from program variables to program values. Metavariable `E` ranges over environments. We use `E(x)`$\uparrow$ to indicate that `E` has no mapping for `x`. When convenient we also view an environment as a set of (program variable, program value) pairs. For each operator `o` in the language we assume the existence of a corresponding semantic function denoted $[\![\texttt{o}]\!]$.

[2] With this our solver syntax is related to syntax defined in the SMTlib standard [2]. It defines a syntax for constraints based on S-expressions, but leaves it up to the solver to provide a meaning to those symbols. Since we treat the solver as a black box, we do not assume anything about the meaning of the symbols

$\boxed{\mathtt{E} \vdash \mathtt{e} \Downarrow \mathtt{v}}$

"Expression e evaluates to value v in the context of environment E."

$$\mathtt{E} \vdash \mathtt{c} \Downarrow \mathtt{c} \qquad \text{(E-Const)}$$

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

$$\frac{\mathtt{E(x)}\uparrow}{\mathtt{E} \vdash \mathtt{x} \Downarrow \mathtt{nil}} \qquad \text{(E-Undef)}$$

$$\frac{\mathtt{E(x)} = \mathtt{v}}{\mathtt{E} \vdash \mathtt{x?} \Downarrow \mathtt{v}} \qquad \text{(E-ReadOnly)}$$

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

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Downarrow \mathtt{v}_1 \qquad \mathtt{E} \vdash \mathtt{e}_2 \Downarrow \mathtt{v}_2 \qquad \mathtt{v}_1 \ [\![\oslash]\!] \ \mathtt{v}_2 = \mathtt{v}}{\mathtt{E} \vdash \mathtt{e}_1 \ \oslash \ \mathtt{e}_2 \Downarrow \mathtt{v}} \qquad \text{(E-Compare)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Downarrow \mathtt{v}_1 \qquad \mathtt{E} \vdash \mathtt{e}_2 \Downarrow \mathtt{v}_2 \qquad \mathtt{v}_1 \ [\![\otimes]\!] \ \mathtt{v}_2 = \mathtt{v}}{\mathtt{E} \vdash \mathtt{e}_1 \ \otimes \ \mathtt{e}_2 \Downarrow \mathtt{v}} \qquad \text{(E-Combine)}$$

$\boxed{M \models C}$

"Mapping M is a *model* of constraint C."

This judgment represents a call to the constraint solver, which we treat as a black box. Here metavariable M ranges over finite mappings from constraint variables x to values v. When convenient we also view a mapping as a set of (variable, value) pairs.The proposition $M \models C$ denotes that M is a solution to the constraint C (and further one that is optimal according to the solver's semantics, as discussed earlier).

We use the notation $\not\models C$ to mean that there is no mapping M such that $M \models C$, i.e., the solver is unable to find a solution, either because C is unsatisfiable or because it is satisfiable but the solver is unable to find one.

$\boxed{\mathtt{E} \vdash \mathtt{e} \rightsquigarrow e}$

$\boxed{\mathtt{E} \vdash \mathtt{C} \rightsquigarrow C}$

We define $\rightsquigarrow$ as a set of functions to translate source-level constraints into the language of the solver. For simplicity and readability, we have defined both languages to use equivalent syntax for some constructs except for font. However, these entities represent different concepts, and, as mentioned in Section 4.6, we do not assume anything about the meaning the solver assigns to programs in the solver language.

The following judgments define how to translate source-level constraints into the language of the solver. For Babelsberg/Reals and Babelsberg/PrimitiveTypes this is straightforward.

$$\frac{\mathtt{c} \neq \mathtt{nil}}{\mathtt{E} \vdash \mathtt{c} \rightsquigarrow c} \quad \text{(ToConst)}$$

$$\mathtt{E} \vdash \mathtt{x} \rightsquigarrow x \quad \text{(ToVar)}$$

$$\frac{\mathtt{E} \vdash \mathtt{x} \rightsquigarrow x}{\mathtt{E} \vdash \mathtt{x?} \rightsquigarrow x?} \quad \text{(ToReadOnly)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \rightsquigarrow e_1 \qquad \mathtt{E} \vdash \mathtt{e}_2 \rightsquigarrow e_2}{\mathtt{E} \vdash \mathtt{e}_1 \ \oplus \ \mathtt{e}_2 \rightsquigarrow e_1 \oplus e_2} \quad \text{(ToExpression)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \rightsquigarrow e_1 \qquad \mathtt{E} \vdash \mathtt{e}_2 \rightsquigarrow e_2}{\mathtt{E} \vdash \mathtt{e}_1 \ \circledcirc \ \mathtt{e}_2 \rightsquigarrow e_1 \circledcirc e_2} \quad \text{(ToAtomicConstraint)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \rightsquigarrow e_1 \qquad \mathtt{E} \vdash \mathtt{e}_2 \rightsquigarrow e_2}{\mathtt{E} \vdash \mathtt{e}_1 \ \circledtriangledown \ \mathtt{e}_2 \rightsquigarrow e_1 \circledtriangledown e_2} \quad \text{(ToConstraint)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e} \rightsquigarrow e}{\mathtt{E} \vdash \rho \ \mathtt{e} \rightsquigarrow \rho \ e} \quad \text{(ToPriority)}$$

$\boxed{M \rightsquigarrow \mathtt{E}}$

This judgment defines how to translate models produced by the constraint solver back into a program environment. For Babelsberg/Reals and Babelsberg/PrimitiveTypes this is straightforward.

$$\emptyset \rightsquigarrow \emptyset \quad \text{(FromEmpty)}$$

$$\frac{M(x) = v \qquad M_0 = M \setminus \{(x, v)\} \qquad M_0 \rightsquigarrow \mathtt{E}_0 \qquad \mathtt{E} = \mathtt{E}_0 \cup \{(\mathtt{x}, \mathtt{v})\}}{M \rightsquigarrow \mathtt{E}} \quad \text{(FromOne)}$$

$\boxed{\mathtt{E} \models_{\mathtt{E}} \mathtt{C}}$

This judgment solves program constraints by translating them to the language of the constraint solver, obtaining a model, and translating back to a program environment.

$$\frac{\mathtt{E} \vdash \mathtt{C} \rightsquigarrow C \qquad M \models C \qquad M \rightsquigarrow \mathtt{E}_s}{\mathtt{E}_s \models_{\mathtt{E}} \mathtt{C}} \quad \text{(Solve)}$$

$\boxed{\not\models_{\mathtt{E}} \mathtt{C}}$

This judgment determines when a program constraint is unsatisfiable or the solver is unable to find a solution.

$$\frac{\mathtt{E} \vdash \mathtt{C} \rightsquigarrow C \qquad \not\models C}{\not\models_{\mathtt{E}} \mathtt{C}} \quad \text{(Fail)}$$

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

This judgment defines how to translate an environment into a source-level "stay" constraint.

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

$$\frac{\texttt{E(x)} = \texttt{v} \qquad \texttt{E}_0 = \texttt{E} \setminus \{(\texttt{x}, \texttt{v})\} \qquad \text{stay}(\texttt{E}_0) = \texttt{C}_0 \qquad \texttt{C} = \texttt{C}_0 \wedge \texttt{weak x=v}}{\text{stay}(\texttt{E}) = \texttt{C}} \qquad \text{(StayOne)}$$

We provide judgments to support the notion of different types for expressions. These provide a non-standard form of type-checking to check whether a constraint expression is valid (i.e., it returns a boolean.) It is non-standard because it only happens dynamically on a constraint, in the context of the current run-time environment. The types are also nonstandard: we only distinguish between booleans, undefined values, and lump all other primitive types together.

$\boxed{\texttt{E} \vdash \texttt{e} : \texttt{T}}$

$\boxed{\texttt{E} \vdash \texttt{C}}$

"Expression `e` has type `T` in the context of environment `E`."

"Constraint `C` is well formed in the context of environment `E`."

We use the following syntax for types.

Type `T` ::= `Boolean` | `Undefined` | `PrimitiveType`

$$\texttt{E} \vdash \texttt{nil} : \texttt{Undefined} \qquad \text{(T-Nil)}$$

$$\texttt{E} \vdash \texttt{true} : \texttt{Boolean} \qquad \text{(T-True)}$$

$$\texttt{E} \vdash \texttt{false} : \texttt{Boolean} \qquad \text{(T-False)}$$

$$\frac{\texttt{c} \notin \{\texttt{true, false, nil}\}}{\texttt{E} \vdash \texttt{c} : \texttt{PrimitiveType}} \qquad \text{(T-Constant)}$$

$$\frac{\texttt{E} \vdash \texttt{x} \Downarrow \texttt{v} \qquad \texttt{E} \vdash \texttt{v} : \texttt{T}}{\texttt{E} \vdash \texttt{x} : \texttt{T}} \qquad \text{(T-Variable)}$$

$$\frac{\texttt{E} \vdash \texttt{x} : \texttt{T}}{\texttt{E} \vdash \texttt{x?} : \texttt{T}} \qquad \text{(T-ReadOnly)}$$

$$\frac{\texttt{E} \vdash \texttt{e}_1 : \texttt{T} \qquad \texttt{E} \vdash \texttt{e}_2 : \texttt{T}}{\texttt{E} \vdash \texttt{e}_1 \oplus \texttt{e}_2 : \texttt{T}} \qquad \text{(T-Op)}$$

$$\frac{\texttt{E} \vdash \texttt{e}_1 : \texttt{T} \qquad \texttt{E} \vdash \texttt{e}_2 : \texttt{T}}{\texttt{E} \vdash \texttt{e}_1 \circledcirc \texttt{e}_2 : \texttt{Boolean}} \qquad \text{(T-Compare)}$$

$$\frac{\texttt{E} \vdash \texttt{e}_1 : \texttt{Boolean} \qquad \texttt{E} \vdash \texttt{e}_2 : \texttt{Boolean}}{\texttt{E} \vdash \texttt{e}_1 \bigtriangledown \texttt{e}_2 : \texttt{Boolean}} \qquad \text{(T-Combine)}$$

$$\frac{\texttt{E} \vdash \texttt{e} : \texttt{T}}{\texttt{E} \vdash \rho\ \texttt{e} : \texttt{T}} \quad \text{(T-Priority)}$$

The following judgment is used to decide which expressions are valid as constraints.

$$\frac{\texttt{E} \vdash \texttt{C} : \texttt{Boolean}}{\texttt{E} \vdash \texttt{C}} \quad \text{(T-Constraint)}$$

The above rule encodes the requirement that expressions used in constraints must evaluate to a boolean – the constraint is that the expression evaluate to `true`.

$$\frac{\texttt{E} \vdash \texttt{x} : \texttt{Undefined}}{\texttt{E} \vdash \texttt{x = e}} \quad \text{(T-Asgn)}$$

$$\texttt{E} \vdash \texttt{x = nil} \quad \text{(T-Undef)}$$

The above two rules are key to allow assigning new variables and to undefine variables. Changing the type of a variable can thus be accomplished by first undefining it, and then assigning a new value.

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

"Configuration `<E|C|s>` takes one step of execution to state `<E'|C'|s'>`."

A "configuration" defining the state of an execution includes a concrete context, represented by the environment, a symbolic context, represented by the constraint, and a statement to be executed. The environment and statement are standard, while the constraint is not part of the state of a computation in most languages. Intuitively, the environment comes from constraint solving during the evaluation of the immediately preceding statement, and the constraint records the `always` constraints that have been declared so far during execution. Also, assignment is just syntactic sugar for one or more `once` constraints. As explained in the examples above, this retains the same outward appearance as normal imperative assignment.

Constraints are typed. Since assignments are converted into `once` constraints, the left hand side must have a compatible type. To ensure that, all variables are first undefined, and then assigned.

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{v}}{\texttt{<E|C|x := e>} \longrightarrow \texttt{<E|C|once x = nil; once x = v>}} \quad \text{(S-Asgn)}$$

$$\frac{\texttt{E} \vdash \texttt{C}_0 \qquad \text{stay}(\texttt{E}) = \texttt{C}_s \qquad \texttt{E}' \models_{\texttt{E}} (\texttt{C} \wedge \texttt{C}_s \wedge \texttt{C}_0)}{\texttt{<E|C|once C}_0\texttt{>} \longrightarrow \texttt{<E'|C|skip>}} \quad \text{(S-Once)}$$

$$\frac{\texttt{E} \vdash \texttt{C}_0 \qquad \text{stay}(\texttt{E}) = \texttt{C}_s \qquad \not\models_{\texttt{E}} (\texttt{C} \wedge \texttt{C}_s \wedge \texttt{C}_0)}{\texttt{<E|C|once C}_0\texttt{>} \longrightarrow \texttt{<E|C|fail>}} \quad \text{(S-OnceFail)}$$

$$\frac{\texttt{E} \vdash \texttt{C}_0 \qquad \texttt{C}' = (\texttt{C} \wedge \texttt{C}_0) \qquad \text{stay}(\texttt{E}) = \texttt{C}_s \qquad \texttt{E}' \models_{\texttt{E}} (\texttt{C}' \wedge \texttt{C}_s)}{\texttt{<E|C|always C}_0\texttt{>} \longrightarrow \texttt{<E'|C'|skip>}} \quad \text{(S-Always)}$$

$$\frac{\texttt{E} \vdash \texttt{C}_0 \qquad \texttt{C}' = (\texttt{C} \wedge \texttt{C}_0) \qquad \text{stay}(\texttt{E}) = \texttt{C}_s \qquad \not\models_{\texttt{E}} (\texttt{C}' \wedge \texttt{C}_s)}{\texttt{<E|C|always C}_0\texttt{>} \longrightarrow \texttt{<E|C|fail>}} \quad \text{(S-AlwaysFail)}$$

$$\texttt{<E|C|skip;}s_2\texttt{>} \longrightarrow \texttt{<E|C|}s_2\texttt{>} \qquad \text{(S-SeqSkip)}$$

$$\frac{\texttt{<E|C|}s_1\texttt{>} \longrightarrow \texttt{<E'|C'|}s_1'\texttt{>}}{\texttt{<E|C|}s_1\texttt{;}s_2\texttt{>} \longrightarrow \texttt{<E'|C'|}s_1'\texttt{;}s_2\texttt{>}} \qquad \text{(S-SeqStep)}$$

$$\texttt{<E|C|fail;}s_2\texttt{>} \longrightarrow \texttt{<E|C|fail>} \qquad \text{(S-SeqFail)}$$

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{true}}{\texttt{<E|C|if e then }s_1\texttt{ else }s_2\texttt{>} \longrightarrow \texttt{<E|C|}s_1\texttt{>}} \qquad \text{(S-IfThen)}$$

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{false}}{\texttt{<E|C|if e then }s_1\texttt{ else }s_2\texttt{>} \longrightarrow \texttt{<E|C|}s_2\texttt{>}} \qquad \text{(S-IfElse)}$$

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{true}}{\texttt{<E|C|while e do s>} \longrightarrow \texttt{<E|C|s; while e do s>}} \qquad \text{(S-WhileDo)}$$

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

5 Babelsberg/Records

For this next language, we augment Babelsberg/PrimitiveTypes with immutable records.

Records are written as lists of name/value pairs in curly braces:

```
{x:5, y:10}
```

For simplicity, we restrict the records to be flat, i.e., the fields can only contain primitives and not other records. (Defining the semantics for recursive record structures turns out to be easier once we have references, so we postpone supporting this until the Babelsberg/UID language. Allowing recursive records in Babelsberg/Records is possible, but makes the formalism for mapping from values in the solver language back to values in the environment more complex, requiring recursive rules, and seems to add little value to our incremental exposition.)

There is no notion of object identity for Babelsberg/Records — we can test whether two records are equal, but whether or not they are identical would be an implementation issue and not part of the semantics.

The syntax is extended to include record constructors and field access. Here are examples of expressions, assignment statements, and constraints involving records:

```
p := {x:2, y:5};   /* assign a record to p */
a := p.x;          /* access a field of a record and assign it to a variable */
q := p;            /* q is now a copy of p (or maybe it's shared; we can't tell
                      the difference) */
always p.x = 100;  /* a constraint on a record field */
always r = p;      /* an equality constraint between two records */
```

There is no static type checking — checking is all done dynamically. Our model includes structural compatibility checks on constraints. These allow us to desugar all of the record constraints (recursively) into constraints on the individual components. Thus, the record constructs are all handled by the semantic rules, and the constraint solver does not need to know anything about records. The alternative of having the solver itself know about records is more complex and seems to lead to more indeterminism, as well as some odd cases (including requiring that the constraint solver sometimes spontaneously generate records with additional fields). There are approaches that accomplish this, for example in BackTalk [13]. For Babelsberg we did not select this alternative due to the added complexity; but for completeness, we outline it in Appendix A.1.

The structural compatibility checks are assertions that are checked before desugaring the constructs involving records, for example, checking whether a variable holds a record, and whether the record has the necessary fields. While these assertions are checked, unlike normal constraints the system will never change anything to enforce them — if one is violated it's just an error. Instead, the programmer must ensure that a record with the expected fields is first assigned to a variable used in record constraints, just as a programmer would need to ensure that a record with the expected fields was assigned to a record-valued variable in a standard language. The check also succeeds for a constraint `p=q` where `p` is uninitialized and `q` holds a value, or vice versa. This is on the one hand convenient, but is actually also essential, because we convert assignments into `once` equality constraints and otherwise there would be no way to initialize a variable.

Here are a few examples.

```
p := {x:2, y:5};
always p.x = 100;
```

After executing the two statements, the environment is $\texttt{p} \mapsto \texttt{\{x:100, y:5\}}$. Here is how the system handles the `always` constraint. The structural compatibility check is that `p` is a record that has an `x` field. We desugar the record constraint, so that after evaluating the `always p.x = 100` statement the constraints are:

weak	$p_x = 2$
weak	$p_y = 5$
required	$p_x = 100$

where p_x and p_y are new variables.

These constraints have the solution $p_x = 100$, $p_y = 5$, which is then reassembled to yield the environment $\texttt{p} \mapsto \texttt{\{x:100, y:5\}}$.

The following program fails the structural compatability checks, since the `always` constraint expects `p` to have a `y` field but it doesn't:

```
p := {x:2};
always p.y = 100;
```

This program fails as well, since constraining a record to be equal to a number fails the compatability check:

```
p := {x:2};
always p = 5;
```

The following program passes the structural compatibility checks, but fails with an unsatisfiable constraint error, since we are requiring that `p.x` be both 100 and 2:

```
p := {x:0, y:0};
always p.x = 100;
p := {x:2, y:5};
```

However, the following program is OK:

```
p := {x:0};
always weak p = {x:3};
always weak p = {x:4}
```

This desugars to the constraints weak $p_x = 3$ and weak $p_x = 4$, plus weak $p_x = 0$ representing the stay constraint. Using the LPB comparator, there are three possible solutions for p_x, which would yield the three different environments `p` $\mapsto$ `{x:0}`, `p` $\mapsto$ `{x:3}`, or `p` $\mapsto$ `{x:4}`. A WSB comparator would allow x to take on any value in $[0, 4]$, although Cassowary would produce either 1, 3 or 4 (but not something in between), i.e., the same solutions as LPB. (If we made the `always` constraints slightly higher priority than the stay, the solutions would be either `p` $\mapsto$ `{x:3}` or `p` $\mapsto$ `{x:4}`.)

Here's another example that fails a structural compatibility check:

```
a := {x:0};
b := {y:0};
always weak a.x = 3
always weak b.y = 10
always weak a=b
```

This is the case even though there actually are records that would satisfy all of the weak constraints. Thus, if the initial assignments had been different:

```
a := {x:0, y:0};
b := {x:0, y:0};
always weak a.x = 3
always weak b.y = 10
always weak a=b
```

the program works.

5.1 Formalism

5.1.1 Syntax

The syntax from Section 4.4 is augmented now to support records and the ability to access fields of a record:

Expression	`e`	`::=`	$\cdots$ | $\{$`l`$_1$`:e`$_1$`,...,l`$_n$`:e`$_n\}$ | `e.l`
Label	`l`	`::=`	record label names

The language of the solver is unchanged. Specifically, the solver does not "understand" records, but only values of base types like integers and reals.

5.1.2 Semantics

For simplicity we only provide the semantics for a subset of the syntax shown above. Specifically, we assume that records cannot be nested within other records, and we assume that records are always assigned a variable name before being used in a constraint. These restrictions simplify the presentation but are not fundamental.

$\boxed{\mathtt{E} \vdash \mathtt{e} \Downarrow \mathtt{v}}$

Expression evaluation is updated to support records. The typing rules ensure that arbitrary predicates and operations cannot be used with records. We add rules to evaluate record expressions, field expressions, and equality between records.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Downarrow \mathtt{v}_1 \cdots \mathtt{E} \vdash \mathtt{e}_n \Downarrow \mathtt{v}_n}{\mathtt{E} \vdash \{\mathtt{l}_1\mathtt{:e}_1,\ldots,\mathtt{l}_n\mathtt{:e}_n\} \Downarrow \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\}} \quad \text{(E-Rec)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e} \Downarrow \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\} \qquad 1 \leq \mathtt{i} \leq \mathtt{n}}{\mathtt{E} \vdash \mathtt{e.l}_i \Downarrow \mathtt{v}_i} \quad \text{(E-Field)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 \Downarrow \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\} \qquad \mathtt{E} \vdash \mathtt{e}_2 \Downarrow \{\mathtt{l}_1\mathtt{:v}_{n+1},\ldots,\mathtt{l}_n\mathtt{:v}_{2n}\} \qquad \mathtt{E} \vdash (\mathtt{v}_1\mathtt{=}\mathtt{v}_{n+1}\wedge\ldots\wedge\mathtt{v}_n\mathtt{=}\mathtt{v}_{2n}) \Downarrow \mathtt{v}}{\mathtt{E} \vdash \mathtt{e}_1 \ \mathtt{=}\ \mathtt{e}_2 \Downarrow \mathtt{v}}$$

(E-CompareRec)

$\boxed{M \models C}$

This judgment represents the solver, as before.

$\boxed{\mathtt{E,H} \vdash \mathtt{e} \rightsquigarrow e}$

$\boxed{\mathtt{E,H} \vdash \mathtt{C} \rightsquigarrow C}$

We extend the judgments to translate source-level constraints into the language of the solver. Now constraints can refer to records, record operations, and variables which refer to records and operations on them. These must get "exploded" into constraints on the individual components of the records. The rules below also enforce the compatibilty checks required to translate to the solver.

$$\mathtt{E} \vdash \mathtt{x.l}_i \rightsquigarrow x_{l_i} \quad \text{(ToFieldVar)}$$

$$\frac{\begin{array}{c}\mathtt{E} \vdash \mathtt{x} \Downarrow \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\} \qquad \mathtt{E} \vdash \mathtt{x}' \Downarrow \{\mathtt{l}_1\mathtt{:v}'_1,\ldots,\mathtt{l}_n\mathtt{:v}'_n\} \\ \mathtt{E} \vdash \mathtt{x.l}_1 \rightsquigarrow e_1,\ldots,\mathtt{E} \vdash \mathtt{x.l}_n \rightsquigarrow e_n \qquad \mathtt{E} \vdash \mathtt{x}'\mathtt{.l}_1 \rightsquigarrow e_{n+1},\ldots,\mathtt{E} \vdash \mathtt{x}'\mathtt{.l}_n \rightsquigarrow e_{2n}\end{array}}{\mathtt{E} \vdash \mathtt{x} \ \mathtt{=}\ \mathtt{x}' \rightsquigarrow e_1 = e_{n+1} \wedge \ldots \wedge e_n = e_{2n}} \quad \text{(ToCompareRec)}$$

$\boxed{M \rightsquigarrow \mathtt{E}}$

We adapt the translation back from the model to support records. We introduce rules to convert variables from the solver into records in the environment.

$$\frac{M(x_{l_i}) = v_i \qquad M_0 = M \setminus \{(x_{l_i}, v_i)\} \qquad M_0 \rightsquigarrow \mathtt{E}_0 \qquad \mathtt{E}_0\mathtt{(x)} = \emptyset \qquad \mathtt{E{=}E}_0 \cup \{\mathtt{(x,\{}\mathtt{l}_i\mathtt{:v}_i\mathtt{\})\}}}{M \rightsquigarrow \mathtt{E}} \quad \text{(FromField)}$$

$$\frac{M(x_{l_{n+1}}) = v_{n+1} \qquad M_0 = M \setminus \{(x_{l_{n+1}}, v_{n+1})\} \qquad M_0 \rightsquigarrow \mathtt{E}_0 \qquad \mathtt{E}_0\mathtt{(x)} = \mathtt{v}_x \qquad \mathtt{v}_x = \mathtt{\{}\mathtt{l}_1\mathtt{:v}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:v}_n\mathtt{\}} \qquad \mathtt{E{=}(E}_0 \setminus \mathtt{\{(x,\ v}_x\mathtt{)\})} \cup \mathtt{\{(x,\{}\mathtt{l}_1\mathtt{:v}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:v}_n\mathtt{,}\mathtt{l}_{n+1}\mathtt{:v}_{n+1}\mathtt{\})\}}}{M \rightsquigarrow \mathtt{E}} \quad \text{(FromFieldMerge)}$$

$\boxed{\mathtt{E} \vdash \mathtt{e} : \mathtt{T}}$

$\boxed{\mathtt{E} \vdash \mathtt{C}}$

We extend our type semantics to support records. As before, we use our typing rules only to check the validity of constraints. For the types we now distinguish between booleans, undefined values, all other primitive types, and record types. The typechecking for records performs a structural compatibility check, so for example only records with the same structure can be used in an equality constraint. We adapt the syntax for types as follows:

Type	T	::=	B | {$\mathtt{l}_1$:B,...,$\mathtt{l}_n$:B}
BaseType	B	::=	Boolean | Undefined | PrimitiveType

Records only support equality between them. To ensure this, our existing typing rules have to change to allow only BaseType operands for $\oplus$ and $\circledcirc$.[3] We add rules to type records, their fields, and equality between records.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 : \mathtt{B}_1 \cdots \mathtt{E} \vdash \mathtt{e}_n : \mathtt{B}_n}{\mathtt{E} \vdash \mathtt{\{}\mathtt{l}_1\mathtt{:e}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:e}_n\mathtt{\}} : \mathtt{\{}\mathtt{l}_1\mathtt{:B}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:B}_n\mathtt{\}}} \quad \text{(T-Rec)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e} : \mathtt{\{}\mathtt{l}_1\mathtt{:B}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:B}_n\mathtt{\}} \qquad 1 \le i \le n}{\mathtt{E} \vdash \mathtt{e.}\mathtt{l}_i : \mathtt{B}_n} \quad \text{(T-Field)}$$

The rule above is important to ensure that a field of a record can only be referred to if it already exists in that record.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 : \mathtt{B} \qquad \mathtt{E} \vdash \mathtt{e}_2 : \mathtt{B}}{\mathtt{E} \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 : \mathtt{B}} \quad \text{(T-Op)}$$

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 : \mathtt{B} \qquad \mathtt{E} \vdash \mathtt{e}_2 : \mathtt{B}}{\mathtt{E} \vdash \mathtt{e}_1 \circledcirc \mathtt{e}_2 : \mathtt{Boolean}} \quad \text{(T-Compare)}$$

The above two rules ensure that arbitrary operations and comparisons only work on primitive types.

$$\frac{\mathtt{E} \vdash \mathtt{e}_1 : \mathtt{\{}\mathtt{l}_1\mathtt{:B}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:B}_n\mathtt{\}} \qquad \mathtt{E} \vdash \mathtt{e}_2 : \mathtt{\{}\mathtt{l}_1\mathtt{:B}_1\mathtt{,\ldots,}\mathtt{l}_n\mathtt{:B}_n\mathtt{\}}}{\mathtt{E} \vdash \mathtt{e}_1 \mathtt{ = } \mathtt{e}_2 : \mathtt{Boolean}} \quad \text{(T-CompareRec)}$$

[3] We could also have record operations ranged over by one or more meta-variables, for example, to do pair-wise arithmetic on records. We omit this since it isn't essential for elucidating the Babelsberg semantics.

<E|C|s> ⟶ <E′|C′|s′>

The semantics for executing statements is identical to what we had before. Importantly, our typechecks from Babelsberg/PrimitiveTypes still apply. Note that we implicitly get stuck if a compatibility check fails. In a practical language, this could be extended to generate an exception instead.

5.2 Adding Mutable Records

It would be easy to extend Babelsberg/Records to allow mutable records. Syntactically, we would simply allow field acccesses as l-values, e.g.,

```
p := {x:0, y:0};
p.x := 100;
```

After the second assignment, this desugars to the constraints weak $p_x = 0$, weak $p_y = 0$, required $p_x = 100$, which after solving and putting the record back together results in the environment `p` $\mapsto$ `{x:100, y:0}`. This doesn't add any particular complications to the semantics.

However, since we translate assignment statements into "once" constraints, the only additional feature provided by such an extension would be the syntax allowing field accesses as l-values — we can always convert such a program into one that only has immutable records. For example, the above program is equivalent to the following program in Babelsberg/Records:

```
p := {x:0, y:0};
once p.x = 100;
```

In any case there still would be no notion of object identity. Consider:

```
p := {x:0, y:0};
q := p;
p.x := 100;
```

After executing the three statements `p` $\mapsto$ `{x:100, y:0}`, but `q` $\mapsto$ `{x:0, y:0}`.

Thus, given that adding mutable records would simply provide an additional syntactic convenience, and given that Babelsberg/Records is just a step toward objects in any case, we don't include this as a separate language.

6 Babelsberg/UID

Babelsberg/UID adds a number of features to the language. As another step toward representing objects, we augment Babelsberg/PrimitiveTypes with records that live on the heap, and that are mutable, have an identity, and can be aliased. As with Babelsberg/Records these are represented as lists of name/value pairs in curly braces:

```
{x:5, y:10}
```

However, to emphasize that we now allocate records on the heap, we use the keyword `new` when creating one. The syntax includes field accesses as both r-values and l-values.

Primitive types, however, still don't live on the heap. (It would be possible to store all data on the heap, but we elected not to, since having everything on the heap makes the descriptions more complex — there would always be a level of indirection to get to data. If one wants the effect of storing an integer or boolean on the heap, it is easy to simulate this by constructing a record that has a single field that holds the integer or boolean.)

Record fields hold either an instance of a primitive type or a reference to another record. There is no syntax for creating a nested record directly — nested records have to be constructed with references, so each record *must* have a variable that refers to it, even if it is only used in a nested structure. This simplifies the semantics. (Nested records could be supported directly with just a source code transformation.)

Since object identity is now significant, we add identity constraints to the language (following Smalltalk syntax, written `==`, in contrast to `=` for equality constraints). Assignment is now translated to a `once` identity constraint. For records `p` and `q`, if `p` and `q` are identical, they must also be equal, but the converse is not necessarily true — if `p` and `q` are equal, they might or might not be identical.

Unlike Babelsberg/PrimitiveTypes, the weak stays on variables are identity constraints rather than equality constraints. For records, such a variable continues to refer to the same object unless reassigned. This is a direct consequence of the weak stays referring to the references the variables hold, not the records on the heap.

For simplicity, we also allow identity tests and constraints on primitive types. (Otherwise we would need two different translations for assignment.) Again for simplicity, two instances of a primitive type are identical iff they are equal. (Allowing them to be equal but not identical would seem to imply storing them on the heap, which complicates the semantics.)

Here are some examples involving record fields and constraints:

```
p := new {x:2, y:5}; /* create a record on the heap and save the reference in p */
a := p.x;            /* read a field of a record and assign it to a variable */
p.x := 6;            /* assign 6 to the field p.x */
always p.x = 100;
```

As with Babelsberg/Records, there is no static type checking — checking is all done dynamically. Again, our model includes structural compatibility checks on constraints, which allows us to desugar all of the record constraints (recursively) into constraints on the individual components, so that the record constructs are all handled by the semantic rules, and the constraint solver does not need to know anything about records.

After executing the first statement, the above example desugars into the following constraints (where r is a reference):

required	$p = r$
required	$r_x = 2$
required	$r_y = 5$

The reference r is a constant, but r_x is a variable representing the `x` field of the record that r points to, and similarly for r_y. After the second statement we have the stay constraints on `p` and its fields, and the required constraint resulting from the assignment:

weak	$p = r$
weak	$r_x = 2$
weak	$r_y = 5$
required	$a = 5$

After the third statement:

weak	$p = r$
weak	$r_x = 2$
weak	$r_y = 5$
weak	$a = 5$
required	$r_x = 6$

And finally:

weak	$p = r$
weak	$r_x = 6$
weak	$r_y = 5$
weak	$a = 5$
required	$r_x = 100$

After solving the final set of constraints and putting the record back together, we have the environment `p` $\mapsto$ `r` $\wedge$ `a` $\mapsto$ `5`, as well as the heap `r` $\mapsto$ `{x:100, y:5}`.

Here are a few examples that fail the structural compatibility checks.

```
p := new {x:2, y:5};
always p.z = 5;
```

As before, the solver is not allowed to add fields to records, so `p` would be required to have a `z` field already.

Variables that refer to records refer only to references on the heap. The following example demonstrates one form of aliasing:

```
p := new {x:2, y:5};
always q == p;
p.x := 100;
```

Now, `q` holds the same reference as `p`, so both will always point to the same record.

The more common form of aliasing is this:

```
p := new {x:2, y:5};
q := p;
p.x := 100;
```

Now `p` and `q` are aliased, so after the program runs we have `p` $\mapsto$ `r` and `q` $\mapsto$ `r` (the same `r` in both mappings), where `r` is a reference that points to the record value `{x:100,y:0}` in the heap. Note that in the latter case (as is known from imperative programs), the aliasing is only temporary and might be broken later.

Another example:

```
p := new {x:0, y:0};
always p == q;
q := new {a:10};
```

Now both `p` and `q` refer to the same record `{a:10}`. Note that we don't need the structural compatibility checks for identity constraints. For example, it's OK that `q` doesn't have a value before evaluating `always p==q`; and we can also change the structure of `p` as well as `q` with the final assignment. (If the always constraint had been a record equality constraint rather than an identity constraint, the assignment to `q` would have failed the structural compatibility check, which would have expected it to be a record with `x` and `y` fields.)

The programmer can annotate identity constraints with priorities:

```
p := new {x:0, y:0};
q := new {z:0};
always p.x = 10;
always q.z = 100;
always p == r;
always weak q == r;
```

Here, the weak identity constraint `always weak q == r` can't be satisfied.

6.1 Formalism

Since this version is significantly different than the prior formalisms, we present it in its entirety rather than as a delta from those ones.

6.1.1 Syntax

The syntax is modified in a few ways, all standard. First, records are not expressions and there are also no record values. Instead records are created via a `new` statement, which allocates the record on the heap and returns a *reference* to it. We introduce a metavariable `r` ranging over references to mutable records, which are a new kind of value in the language. It is convenient to consider references to be expressions, but we assume they are never used in source programs.

Statement	s	::=	`skip` | `a := e` | `x := new o`
		|	`always C` | `once C` | `s;s` | `fail`
		|	`if e then s else s` | `while e do s`
Constraint	C	::=	`e` | ρ `e`
Expression	e	::=	`c` | `a` | `a?` | `r` | `a` $\stackrel{\texttt{r}}{=}$ `a` | `a == a` | `e` $\oplus$ `e` | `e` $\oslash$ `e` | `e` $\oslash$ `e`
Accessor	a	::=	`x` | `a.l`
Constant	c	::=	`true` | `false` | `nil` | base type constants
Variable	x	::=	variable names
Label	l	::=	record label names
Object	o	::=	$\{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}$
Reference	r	::=	references to heap records
Value	v	::=	`c` | `r`

Note that we augment the language with two operations over references, namely logical and pointer equality, respectively denoted $\stackrel{\texttt{r}}{=}$ and `==`. The arguments to these operations are not arbitrary expressions but rather *accessors*, which are essentially access paths through a reference. This is no loss of expressiveness but is useful for the translation to the constraint solver.

6.1.2 Semantics

In addition to an environment, the semantics now requires a *heap*, which is a partial function from mutable references to record "objects" of the form $\{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\}$. Heaps are ranged over by metavariable H. Most of the updates to the original rules are standard and straightforward.

$\boxed{\mathtt{E;H} \vdash \mathtt{e} \Downarrow \mathtt{v}}$

$\boxed{\mathtt{E;H} \vdash \mathtt{a} \Downarrow \mathtt{v}}$

"Expression e evaluates to value v in the context of environment E and heap H."

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

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

$$\frac{\mathtt{E(x)}\uparrow}{\mathtt{E;H} \vdash \mathtt{x} \Downarrow \mathtt{nil}} \qquad \text{(E-Undef)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{a} \Downarrow \mathtt{r} \qquad \mathtt{H(r)} = \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\} \qquad 1 \le \mathrm{i} \le \mathrm{n}}{\mathtt{E;H} \vdash \mathtt{a.l}_i \Downarrow \mathtt{v}_i} \qquad \text{(E-Field)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{a} \Downarrow \mathtt{v}}{\mathtt{E;H} \vdash \mathtt{a?} \Downarrow \mathtt{v}} \qquad \text{(E-ReadOnly)}$$

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

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

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

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

$$\frac{\mathtt{E;H} \vdash \mathtt{a}_1 \Downarrow \mathtt{r}_1 \qquad \mathtt{E;H} \vdash \mathtt{a}_2 \Downarrow \mathtt{r}_2 \qquad \mathtt{H(r}_1\mathtt{)} = \{\mathtt{l}_1\mathtt{:v}_1,\ldots,\mathtt{l}_n\mathtt{:v}_n\} \qquad \mathtt{H(r}_2\mathtt{)} = \{\mathtt{l}_1\mathtt{:v}_{n+1},\ldots,\mathtt{l}_n\mathtt{:v}_{2n}\} \quad \mathtt{E;H} \vdash (\mathtt{v}_1\mathtt{=v}_{n+1} \wedge \ldots \wedge \mathtt{v}_n\mathtt{=v}_{2n}) \Downarrow \mathtt{v}}{\mathtt{E;H} \vdash \mathtt{a}_1 \overset{\mathtt{r}}{=} \mathtt{a}_2 \Downarrow \mathtt{v}} \qquad \text{(E-CompareRec)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{a}_1 \Downarrow \mathtt{r} \qquad \mathtt{E;H} \vdash \mathtt{a}_2 \Downarrow \mathtt{r}}{\mathtt{E;H} \vdash \mathtt{a}_1 \ \mathtt{==} \ \mathtt{a}_2 \Downarrow \mathtt{true}} \qquad \text{(E-CompareRecIDTrue)}$$

$$\frac{\mathtt{E;H} \vdash \mathtt{a}_1 \Downarrow \mathtt{r}_1 \qquad \mathtt{E;H} \vdash \mathtt{a}_2 \Downarrow \mathtt{r}_2 \qquad \mathtt{r}_1 \neq \mathtt{r}_2}{\mathtt{E;H} \vdash \mathtt{a}_1 \ \mathtt{==} \ \mathtt{a}_2 \Downarrow \mathtt{false}} \qquad \text{(E-CompareRecIDFalse)}$$

$\boxed{M \models C}$

This judgment still represents the call to the solver.

$\boxed{\texttt{E,H} \vdash \texttt{e} \rightsquigarrow e}$

$\boxed{\texttt{E,H} \vdash \texttt{C} \rightsquigarrow C}$

$$\texttt{E,H} \vdash \texttt{c} \rightsquigarrow c \qquad \text{(ToConst)}$$

$$\texttt{E,H} \vdash \texttt{r} \rightsquigarrow r \qquad \text{(ToRef)}$$

Above for convenience we consider the solver to have a set of constants ranged over by metavariable r, with each reference from the program mapped to a distinct constant. In practice we can simply use integers.

$$\texttt{E,H} \vdash \texttt{x} \rightsquigarrow x \qquad \text{(ToVar)}$$

$$\frac{\texttt{E,H} \vdash \texttt{a} \rightsquigarrow x}{\texttt{E,H} \vdash \texttt{a.l} \rightsquigarrow x_l} \qquad \text{(ToFieldVar)}$$

The above translation of accessors introduces new variables. For example, the expression `b.f.g` will get translated to the variable x_{f_g}. To relate these variables to the rest of the constraints, we also need to generate a set of axioms that are passed to the solver along with the translated constraint. Specifically, for each variable in the constraint of the form x_f (where x itself may be subscripted multiple times), for each reference `r` in dom(`H`), we generate the following axiom:

$$x = r \Rightarrow x_f = x\_r_f$$

Here x_r_f is a variable introduced by the stay constraints on the heap, defined below. Finally, we also recursively generate such axioms for the variable x itself if it is also subscripted.

$$\frac{\texttt{E,H} \vdash \texttt{a} \rightsquigarrow x}{\texttt{E,H} \vdash \texttt{a?} \rightsquigarrow x?} \qquad \text{(ToReadOnly)}$$

$$\frac{\texttt{E,H} \vdash \texttt{e}_1 \rightsquigarrow e_1 \qquad \texttt{E,H} \vdash \texttt{e}_2 \rightsquigarrow e_2}{\texttt{E,H} \vdash \texttt{e}_1 \ \oplus \ \texttt{e}_2 \rightsquigarrow e_1 \oplus e_2} \qquad \text{(ToExpression)}$$

$$\frac{\texttt{E,H} \vdash \texttt{e}_1 \rightsquigarrow e_1 \qquad \texttt{E,H} \vdash \texttt{e}_2 \rightsquigarrow e_2}{\texttt{E,H} \vdash \texttt{e}_1 \ \circledS \ \texttt{e}_2 \rightsquigarrow e_1 \circledS e_2} \qquad \text{(ToAtomicConstraint)}$$

$$\frac{\texttt{E,H} \vdash \texttt{e}_1 \rightsquigarrow e_1 \qquad \texttt{E,H} \vdash \texttt{e}_2 \rightsquigarrow e_2}{\texttt{E,H} \vdash \texttt{e}_1 \ \circledV \ \texttt{e}_2 \rightsquigarrow e_1 \circledV e_2} \qquad \text{(ToConstraint)}$$

$$\frac{\texttt{E;H} \vdash \texttt{a}_1 \Downarrow \texttt{r}_1 \qquad \texttt{H(r}_1\texttt{)} = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\} \qquad \texttt{E;H} \vdash \texttt{a}_2 \Downarrow \texttt{r}_2 \qquad \texttt{H(r}_2\texttt{)} = \{\texttt{l}_1\texttt{:v}'_1\texttt{,...,l}_n\texttt{:v}'_n\} \quad \texttt{E,H} \vdash \texttt{a}_1\texttt{.l}_1 \texttt{ = a}_2\texttt{.l}_1 \wedge \ldots \wedge \texttt{a}_1\texttt{.l}_n \texttt{ = a}_n\texttt{.l}_n \rightsquigarrow e}{\texttt{E,H} \vdash \texttt{a}_1 \texttt{ = a}_2 \rightsquigarrow e} \qquad \text{(ToCompareRec)}$$

$$\frac{\texttt{E,H} \vdash \texttt{a}_1 \rightsquigarrow x_1 \qquad \texttt{E,H} \vdash \texttt{a}_2 \rightsquigarrow x_2}{\texttt{E,H} \vdash \texttt{a}_1 \texttt{ == } \texttt{a}_2 \rightsquigarrow x_1 = x_2} \quad \text{(ToIdentical)}$$

The above rule shows that identity constraints are just constraints on the direct binding of variables for the solver. We assume that `references` are a finite type that the solver can handle. So if we ask the solver to assign the same identity to $\texttt{x}_1$ and $\texttt{x}_2$, we are just asking that the resulting model assign the same reference to both. For the solver, this is the same kind of equality as for other finite primitive types (e.g. booleans). The idea is that the solver cannot make up references, but rather that references are a finite set that is handed to the solver every time it is invoked (so the solver can only use references that were created using new, and, in a practical language, that have not been garbage collected.)

$$\frac{\texttt{E,H} \vdash \texttt{e} \rightsquigarrow e}{\texttt{E,H} \vdash \rho\ \texttt{e} \rightsquigarrow \rho\ e} \quad \text{(ToPriority)}$$

$\boxed{M \rightsquigarrow \texttt{E,H}}$

$$\emptyset \rightsquigarrow \emptyset,\emptyset \quad \text{(FromEmpty)}$$

$$\frac{M(x) = c \qquad M_0 = M \setminus \{(x,c)\} \qquad M_0 \rightsquigarrow \texttt{E}_0\texttt{,H}_0 \qquad \texttt{E} = \texttt{E}_0 \cup \{(\texttt{x},\texttt{c})\}}{M \rightsquigarrow \texttt{E,H}_0} \quad \text{(FromConst)}$$

$$\frac{M(x) = r \qquad M_0 = M \setminus \{(x,r)\} \qquad M_0 \rightsquigarrow \texttt{E}_0\texttt{,H}_0 \qquad \texttt{E} = \texttt{E}_0 \cup \{(\texttt{x},\texttt{r})\}}{M \rightsquigarrow \texttt{E,H}_0} \quad \text{(FromRef)}$$

$$\frac{M(x\_r_l) = v \qquad M_0 = M \setminus \{(x\_r_l, v)\} \qquad M_0 \rightsquigarrow \texttt{E}_0\texttt{,H}_0 \qquad \texttt{H}_0(\texttt{r}) = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_i\texttt{:v}_i\} \quad \texttt{H} = (\texttt{H}_0 \setminus \{(\texttt{r}, \texttt{H}_0(\texttt{r}))\}) \cup \{(\texttt{r}, \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_i\texttt{:v}_i\texttt{,l:v}\})\}}{M \rightsquigarrow \texttt{E}_0\texttt{,H}} \quad \text{(FromField)}$$

$$\frac{M(x\_r_l) = v \qquad M_0 = M \setminus \{(x\_r_l, v)\} \qquad M_0 \rightsquigarrow \texttt{E}_0\texttt{,H}_0 \qquad \texttt{r} \notin \texttt{dom(H}_0\texttt{)} \quad \texttt{H} = \texttt{H}_0 \cup \{(\texttt{r}, \{\texttt{l:v}\})\}}{M \rightsquigarrow \texttt{E}_0\texttt{,H}} \quad \text{(FromFieldFirst)}$$

$\boxed{\texttt{E;H} \models_{E,H} \texttt{C}}$

$\boxed{\not\models_{E,H} \texttt{C}}$

Analogously to before, these judgments solve program constraints and either translate the solution back to a program environment and heap, or determine that no solution is available.

$$\frac{\texttt{E,H} \vdash \texttt{C} \rightsquigarrow C \qquad M \models C \qquad M \rightsquigarrow \texttt{E}_s\texttt{,H}_s}{\texttt{E}_s\texttt{;H}_s \models_{E,H} \texttt{C}} \quad \text{(Solve)}$$

$$\frac{\texttt{E,H} \vdash \texttt{C} \rightsquigarrow C \qquad \not\models C}{\not\models_{E,H} \texttt{C}} \quad \text{(Fail)}$$

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

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

These judgments define how to translate environment and heap into a source-level "stay" constraint. The rules are unchanged for primitive types that are stored in the environment, but are amended for the heap.

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

$$\frac{\mathtt{E}(\mathtt{x}) = \mathtt{v} \qquad \mathtt{E}_0 = \mathtt{E} \setminus \{(\mathtt{x}, \mathtt{v})\} \qquad \text{stay}(\mathtt{E}_0) = \mathtt{C}_0 \qquad \mathtt{C} = \mathtt{C}_0 \wedge \mathtt{weak\ x{=}v}}{\text{stay}(\mathtt{E}) = \mathtt{C}} \qquad \text{(StayOne)}$$

$$\frac{\mathtt{o} = \{\mathtt{l}_0 : \mathtt{v}_0, \ldots, \mathtt{l}_n : \mathtt{v}_n\} \qquad \mathtt{H}(\mathtt{r}) = \mathtt{o} \qquad \mathtt{H}_0 = \mathtt{H} \setminus \{(\mathtt{r}, \mathtt{o})\} \qquad \text{stay}(\mathtt{H}_0) = \mathtt{C}_0}{\text{stay}(\mathtt{H}) = \mathtt{C}_0 \wedge \mathtt{weak\ x\_r}_{l_0} = \mathtt{v}_0 \wedge \ldots \wedge \mathtt{weak\ x\_r}_{l_n} = \mathtt{v}_n} \qquad \text{(StayRef)}$$

$\boxed{\mathtt{E;H} \models_{E,H} \mathtt{C}}$

"Environment E and heap H are a model of constraint C."

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

$\boxed{\mathtt{E;H} \vdash \mathtt{C} : \mathtt{T}}$

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

"Expression e has type T in the context of environment E and heap H."

"Constraint C has type T in the context of environment E and heap H."

"Constraint C is well formed in the context of environment E and heap H."

We use the following syntax for types.

Type	T	::=	B | | $\{\mathtt{l}_1 : \mathtt{T}_1, \ldots, \mathtt{l}_n : \mathtt{T}_n\}$
BaseType	B	::=	Boolean | Undefined | PrimitiveType

We assume that each operator $\oplus$ has an associated function typeof that provides the types of its arguments and result. Similarly, we assume that each operator $\oslash$ has an associated function typeof that provides the types of its arguments.

$$\mathtt{E;H} \vdash \mathtt{nil} : \mathtt{Undefined} \qquad \text{(T-Nil)}$$

$$\mathtt{E;H} \vdash \mathtt{true} : \mathtt{Boolean} \qquad \text{(T-True)}$$

$$\mathtt{E;H} \vdash \mathtt{false} : \mathtt{Boolean} \qquad \text{(T-False)}$$

$$\frac{\mathtt{c} \notin \{\mathtt{true, false, nil}\}}{\mathtt{E;H} \vdash \mathtt{c} : \mathtt{PrimitiveType}} \qquad \text{(T-Constant)}$$

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

$$\frac{\texttt{E;H} \vdash \texttt{a} \Downarrow \texttt{r} \qquad \texttt{H(r)} = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\} \qquad \texttt{E;H} \vdash \texttt{v}_i : \texttt{T}_i \qquad 1 \leq i \leq n}{\texttt{E;H} \vdash \texttt{x.l}_i : \texttt{T}_i} \quad \text{(T-Field)}$$

$$\frac{\texttt{E;H} \vdash \texttt{a} : \texttt{T}}{\texttt{E;H} \vdash \texttt{a?} : \texttt{T}} \quad \text{(T-ReadOnly)}$$

$$\frac{\texttt{H(r)} = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\} \qquad \texttt{E;H} \vdash \texttt{v}_1 : \texttt{T}_1 \cdots \texttt{E;H} \vdash \texttt{v}_n : \texttt{T}_n}{\texttt{E;H} \vdash \texttt{r} : \{\texttt{l}_1\texttt{:T}_1\texttt{,...,l}_n\texttt{:T}_n\}} \quad \text{(T-Ref)}$$

Note that the above rule does not work as intended in the presence of cyclic records, which are now possible. In that case, we will simply not be able to find a finite derivation and so will raise an error even when the given record satisfies our intended structural compatibility check. This should be fixable by keeping track of references that have already been "visited" in order to know when to stop, and by adding a kind of recursive record type to represent the structure of cyclic records.

$$\frac{\texttt{E;H} \vdash \texttt{a}_1 : \texttt{T} \qquad \texttt{E;H} \vdash \texttt{a}_2 : \texttt{T}}{\texttt{E;H} \vdash \texttt{a}_1 \stackrel{\texttt{r}}{=} \texttt{a}_2 : \texttt{Boolean}} \quad \text{(T-RecEq)}$$

$$\frac{\texttt{E;H} \vdash \texttt{a}_1 : \texttt{T} \qquad \texttt{E;H} \vdash \texttt{a}_2 : \texttt{T}}{\texttt{E;H} \vdash \texttt{a}_1 \texttt{ == } \texttt{a}_2 : \texttt{Boolean}} \quad \text{(T-RecID)}$$

$$\frac{\text{typeof}(\oplus) = (\texttt{B}_1, \texttt{B}_2, \texttt{B}) \qquad \texttt{E;H} \vdash \texttt{e}_1 : \texttt{B}_1 \qquad \texttt{E;H} \vdash \texttt{e}_2 : \texttt{B}_2}{\texttt{E;H} \vdash \texttt{e}_1 \oplus \texttt{e}_2 : \texttt{B}} \quad \text{(T-Op)}$$

$$\frac{\text{typeof}(\circledcirc) = (\texttt{B}_1, \texttt{B}_2) \qquad \texttt{E;H} \vdash \texttt{e}_1 : \texttt{B}_1 \qquad \texttt{E;H} \vdash \texttt{e}_2 : \texttt{B}_2}{\texttt{E;H} \vdash \texttt{e}_1 \circledcirc \texttt{e}_2 : \texttt{Boolean}} \quad \text{(T-Compare)}$$

$$\frac{\texttt{E;H} \vdash \texttt{e}_1 : \texttt{Boolean} \qquad \texttt{E;H} \vdash \texttt{e}_2 : \texttt{Boolean}}{\texttt{E;H} \vdash \texttt{e}_1 \circledtriangledown \texttt{e}_2 : \texttt{Boolean}} \quad \text{(T-Combine)}$$

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

The following judgment is used to decide which expressions are valid as constraints.

$$\frac{\texttt{E;H} \vdash \texttt{C} : \texttt{Boolean}}{\texttt{E} \vdash \texttt{C}} \quad \text{(T-Constraint)}$$

The above rule encodes the requirement that expressions used in constraints must evaluate to a boolean – the constraint is that the expression evaluate to `true`.

$$\frac{\texttt{E;H} \vdash \texttt{a} : \texttt{Undefined} \qquad \texttt{E;H} \vdash \texttt{e} : \texttt{B}}{\texttt{E;H} \vdash \texttt{a = e}} \quad \text{(T-Asgn)}$$

$$\frac{\texttt{E;H} \vdash \texttt{a} : \texttt{Undefined} \qquad \texttt{E;H} \vdash \texttt{e} : \{\texttt{l}_1\texttt{:T}_1\texttt{,...,l}_n\texttt{:T}_n\}}{\texttt{E;H} \vdash \texttt{a == e}} \quad \text{(T-AsgnRef)}$$

$$\texttt{E;H} \vdash \texttt{x = nil} \qquad \text{(T-Undef1)}$$

$$\frac{\texttt{E;H} \vdash \texttt{a} : \texttt{T}}{\texttt{E;H} \vdash \texttt{a = nil}} \qquad \text{(T-Undef2)}$$

The above four rules are key to allow assigning new variables and to undefine variables. Changing the type of a variable can thus be accomplished by first undefining it, and then assigning a new value.

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

"Configuration <E|H|C|s> takes one step of execution to state <E'|H'|C'|s'>."

$$\frac{\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{v} \qquad \texttt{v is not a reference}}{\texttt{<E|H|C|a := e>} \longrightarrow \texttt{<E|H|C|once x = nil; once a = v>}} \qquad \text{(S-Asgn)}$$

$$\frac{\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{r}}{\texttt{<E|H|C|a := e>} \longrightarrow \texttt{<E|H|C|once x = nil; once a == r>}} \qquad \text{(S-AsgnRef)}$$

$$\frac{\begin{array}{c}\texttt{E;H} \vdash \texttt{e}_1 \Downarrow \texttt{v}_1 \cdots \texttt{E;H} \vdash \texttt{e}_n \Downarrow \texttt{v}_n \qquad \texttt{r} \notin \text{dom}(\texttt{H}) \\ \texttt{E';H'} \models_{E,H} \texttt{(C} \wedge \text{stay}(\texttt{E}) \wedge \text{stay}(\texttt{H}) \wedge \texttt{x=r} \wedge \texttt{r = } \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}\texttt{)}\end{array}}{\texttt{<E|H|C|x := new } \{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}\texttt{>} \longrightarrow \texttt{<E'|H'|C|skip>}} \qquad \text{(S-AsgnNew)}$$

The above rule could also be defined through desugaring to a once constraint, but it'd be a bit weirder because we'd have to update the heap first to contain the new reference.

$$\frac{\begin{array}{c}\texttt{E;H} \vdash \texttt{e}_1 \Downarrow \texttt{v}_1 \cdots \texttt{E;H} \vdash \texttt{e}_n \Downarrow \texttt{v}_n \qquad \texttt{r} \notin \text{dom}(\texttt{H}) \\ \not\models_{E,H} \texttt{(C} \wedge \text{stay}(\texttt{E}) \wedge \text{stay}(\texttt{H}) \wedge \texttt{x=r} \wedge \texttt{r = } \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}\texttt{)}\end{array}}{\texttt{<E|H|C|x := new } \{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}\texttt{>} \longrightarrow \texttt{<E|H|C|fail>}} \qquad \text{(S-AsgnNewFail)}$$

$$\frac{\texttt{E;H} \vdash \texttt{C}_0 \qquad \text{stay}(\texttt{E}) = \texttt{C}_e \qquad \text{stay}(\texttt{H}) = \texttt{C}_h \qquad \texttt{E';H'} \models_{E,H} \texttt{(C} \wedge \texttt{C}_0 \wedge \texttt{C}_e \wedge \texttt{C}_h\texttt{)}}{\texttt{<E|H|C|once C}_0\texttt{>} \longrightarrow \texttt{<E'|H'|C|skip>}} \qquad \text{(S-Once)}$$

$$\frac{\texttt{E;H} \vdash \texttt{C}_0 \qquad \text{stay}(\texttt{E}) = \texttt{C}_e \qquad \text{stay}(\texttt{H}) = \texttt{C}_h \qquad \not\models_{E,H} \texttt{(C} \wedge \texttt{C}_0 \wedge \texttt{C}_e \wedge \texttt{C}_h\texttt{)}}{\texttt{<E|H|C|once C}_0\texttt{>} \longrightarrow \texttt{<E|H|C|fail>}} \qquad \text{(S-OnceFail)}$$

$$\frac{\texttt{E;H} \vdash \texttt{C}_0 \qquad \texttt{C}' = \texttt{(C} \wedge \texttt{C}_0\texttt{)} \qquad \text{stay}(\texttt{E}) = \texttt{C}_e \qquad \text{stay}(\texttt{H}) = \texttt{C}_h \qquad \texttt{E';H'} \models_{E,H} \texttt{(C'} \wedge \texttt{C}_e \wedge \texttt{C}_h\texttt{)}}{\texttt{<E|H|C|always C}_0\texttt{>} \longrightarrow \texttt{<E'|H'|C'|skip>}} \qquad \text{(S-Always)}$$

$$\frac{\texttt{E;H} \vdash \texttt{C}_0 \qquad \texttt{C}' = \texttt{(C} \wedge \texttt{C}_0\texttt{)} \qquad \text{stay}(\texttt{E}) = \texttt{C}_e \qquad \text{stay}(\texttt{H}) = \texttt{C}_h \qquad \not\models_{E,H} \texttt{(C'} \wedge \texttt{C}_e \wedge \texttt{C}_h\texttt{)}}{\texttt{<E|H|C|always C}_0\texttt{>} \longrightarrow \texttt{<E|H|C|fail>}} \qquad \text{(S-AlwaysFail)}$$

$$\frac{\texttt{<E|H|C|s}_1\texttt{>} \longrightarrow \texttt{<E'|H'|C'|s}_1'\texttt{>}}{\texttt{<E|H|C|s}_1\texttt{;s}_2\texttt{>} \longrightarrow \texttt{<E'|H'|C'|s}_1'\texttt{;s}_2\texttt{>}} \qquad \text{(S-SeqStep)}$$

$$\texttt{<E|H|C|skip;s}_2\texttt{>} \longrightarrow \texttt{<E|H|C|s}_2\texttt{>} \qquad \text{(S-SeqSkip)}$$

$$\texttt{<E|H|C|fail;s}_2\texttt{>} \longrightarrow \texttt{<E|H|C|fail>} \qquad \text{(S-SeqFail)}$$

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

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

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

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

7 Babelsberg/Objects

In this final language, we add classes, inheritance, methods, and messages. For simplicity, we remove field access entirely — access to an objects variables is only possible through message sends. This language is intended to capture the essential features of actual object constraint languages including Babelsberg/R, Babelsberg/JS, and Babelsberg/Squeak. In particular, it allows us to write constraints on the results of message sends, such as the following constraint on the center of a rectangle:

```
always r.center = Point.new(100,50);
```

The constraint here is an equality constraint on the result of sending the message `center` to `r`, where `center` may be computed rather than being stored. Even the `x` message to a point might compute the x value, if the point is stored in polar coordinates. Further, expressions such as `a+b` are now treated in an object-oriented fashion, so that this means "send the message `+` with the argument `b` to the object `a`," with the meaning of `+` in this expression (and in constraints such as `a+b=c`) depending on the class of `a`.

As with the earlier Babelsbergs presented in this memo, we use a solver language to represent the constraints that are sent to the constraint solvers. Syntactically we will be able to use the same solver language that we defined for Babelsberg/PrimitiveTypes (Section 4.4.1). Semantically, however, the simple translation from source-level constraints into the language of the solver becomes more complex, since we now want to support constraints on objects and in particular on the results of message sends.

To accomplish this, we start with a standard model for an object-oriented language with instances, classes, messages, methods, and inheritance, and add constraints on top of that. Some of the classes and methods in the source language have corresponding primitive types and operations in the solver language, and the semantics includes a partial mapping function to do this conversion. For example, the classes `Integer`, `Float`, and `Boolean` in the source language might be mapped to the types *Integer*, *Float*, and *Boolean* in the solver language. The `+` method on `Integer` would be mapped to the + operation for *Integer* in the solver language, and similarly for the `+` method for `Float`, the `or` method for `Boolean`, and so forth.[4] In general, there will be classes in the source language that don't map to a solver type; and even for a class that does have a corresponding solver type, that class can have additional methods that aren't mapped to operations

[4] As in the previous languages, we use different fonts to distinguish the source and solver languages: `Float` and `x+y` for the class and an expression respectively in the source language; *Float* and $x + y$ for the corresponding type and expression in the solver language. In this example we've used the same names for the classes and types, and methods and operations, but this isn't required — for example the `plus` method might be mapped to the + operator.

in the solver language, for example a `factorial` method for `Integer`. It is also possible to have types and operations in the solver language without equivalents in the source language (although expressions involving them could never be generated).

As with the previous languages, the semantics includes two different execution modes: imperative execution mode and constraint construction mode. For ordinary statements, expressions are evaluated in imperative execution mode. After every statement execution, the current set of constraints is solved and the environment is updated. The current set of constraints is determined as follows. If the statement is an assignment, we evaluate the right hand side in imperative execution mode, and then set up a constraint between the resulting value and the left hand side, as in the previous simpler Babelsberg languages. (As with Babelsberg/UID, and in contrast to the simpler languages, this will be an identity constraint rather than an equality constraint.) In addition, each closure in the constraint store is evaluated to produce a set of constraints in the solver language.[5] The resulting collection of constraints in the solver language is given to the solver. If the solver finds a solution, the environment is updated; but if the constraints have no solution, or if they are too hard for the solver, the system enters the `fail` state.

Also as before, the evaluation of the closures in the constraint store is done in constraint construction mode rather than in imperative execution mode. This is more complex than in the previous languages since constraint expressions might invoke methods. In constraint construction mode, when we encounter a message send that corresponds to an operation on a primitive type in the solver language, we construct the corresponding parse tree fragment for the solver language. Otherwise, we do a normal method lookup and evaluate the body of the method. If we we encounter something we cannot transform (such as a side-effect), we don't transform it, and instead execute. As before, assignment statements are converted to `once` identity constraints with the right-hand-side annotated as read-only.

There are two interpretations or meanings for each constraint at a particular step in the execution of a program in the Babelsberg/Objects semantics: first, saying the constraint is satisfied is equivalent to saying that the result of evaluting the constraint expression in imperative execution mode is `true`; and second, saying the constraint is satisfied is equivalent to saying that the generated constraint in the solver language is satisfied. Ideally, we would be able to prove that these two interpretations are equivalent for all Babelsberg/Objects programs, based on the rules in the formal semantics. Further, the behavior of an interpreter or compiler for an actual Babelsberg language should match this interpretation — the result of evaluating the constraint expression in imperative execution model should give the same result as given by the formal semantics.

7.1 Control Structures and Methods

Babelsberg/Objects has the simple `if` and `while` control structures that were introduced for Babelsberg/PrimitiveTypes. There are no particular complications in including these in Babelsberg/Objects.

In addition, however, Babelsberg/Objects includes methods. Following our overarching design goal of having a standard object-oriented language in the absence of constraints, in imperative execution mode, methods are standard. Suppose we are evaluating `x.m(a,b)`. We first look in `x`'s class for a method with selector `m`, then its superclass, and so on up the superclass chain. If no such method is found, we enter the `fail` state.[6] Otherwise we evaluate the body of the method in a new environment, and return the result. Arguments are

[5] Note that this is just a description of the semantics. In a practical implementation, we would for example only re-evaluate a closure in the constraint store when needed rather than after executing every statement. However, this is just an optimization, which shouldn't change the result of executing the program.

[6] This is analogous to the structural compatibility checks from Babelsberg/Records, and just as in Babelsberg/Records we don't invent a new field for a record if needed to satisfy a constraint, in Babelsberg/Objects we don't convert an object to an instance of a different class, or synthesize a new method, to satisfy constraints — instead, if there is a missing method it's just an error.

passed by value (with pointer semantics), as in most standard object-oriented languages.

Methods can also be invoked by constraint expressions. The correct method is looked up in the standard way, but now the body of the method is evaluated in constraint construction mode rather than imperative execution mode. The arguments are passed by call-by-identity (i.e., by setting up required identity constraints between the actual and formal parameters), rather than call-by-value. The method call in the constraint expression is replaced with a new temporary variable, and the result returned by the method has an identity constraint that it have the same reference as that temporary.

Note that unlike in Babelsberg/R [4], there is no requirement that variables in methods called from constraints be used in a single assignment fashion. (See Section 7.7 for more on this issue.)

7.2 Examples

First, let's revisit the initial example from Babelsberg/Reals, now as a Babelsberg/Objects program:

```
x := 3;
x := 4;
always x>=10;
```

The behavior is quite similar. After evaluating the first statement we hand the following constraint to the solver to trivially find a value for `x`:

required	$x = 3$

The solver finds a value for x, which is then used to update the environment to be $\texttt{x} \mapsto \texttt{3}$. After the second statement we hand the following constraints to the solver:

weak	$x = 3$
required	$x = 4$

This has the solution $x = 4$, resulting in a new environment $\texttt{x} \mapsto \texttt{4}$. Executing the third statement adds the `always` constraint to the current constraint store. We then evaluate the body of the closure in constraint construction mode, resulting in the constraint required $x \geq 10$. We now hand this constraint and the stay constraint to the solver:

weak	$x = 4$
required	$x \geq 10$

With a metric comparator this yields the solution $x = 10$.

Next, consider methods and constraints involving an object with parts and subparts, namely a rectangle represented by storing points that represent the upper left and lower right corners (called `origin` and `corner` in Smalltalk).

Here is a constraint on the upper left corner of a rectangle:

```
always r.upper_left = Point.new(10,10);
```

Next, here is a constraint on the center of a rectangle. The center is a computed value rather than being stored as an instance variable.

```
always r.center = Point.new(100,50);
```

The `center` method for `Rectangle` is defined as follows:

```
def center
  return (self.upper_left + self.lower_right) / 2
end
```

The constraint is evaluated in the following way. We look up the `center` method, and evaluate it in constraint construction mode. This sets up an equality constraint between the result of evaluating `(self.upper_left + self.lower_right) / 2` and a new point with `x=100`, `y=50`. The `=` method for points is then called, resulting in a network of primitive constraints in the solver language:

```
(upper_left.x + lower_right.x) / 2 = 100
(upper_left.y + lower_right.y) / 2 = 50
```

The following example illustrates that methods can also be used with constraints on primitive types.

```
def double(x)
  return 2*x
end;
```

If we use this in a constraint `always 10=double(a)`, the system should satisfy the constraint by making `a` be 5.

Methods can themselves use `once` or `always` constraints (including when they are invoked from other constraint expressions).

7.3 Identity Constraint Examples

The following example illustrates how assignment for objects with object identity is handled.

```
p1 := Point.new(10,10);
p2 := p1;
p1 := Point.new(50,50);
```

The result is just the same as in Ruby or some other object-oriented language without constraints. After the first statement, we have the new point with `x=10,y=10` in the heap, and solve a required identity constraint that `p1`'s value equal the reference to this point. After the second statement, the identity constraints are a weak identity stay on `p1`, and a required constraint that `p2` reference the same value as `p1`. After the third statement, there is another point with `x=50,y=50` in the heap, a required constraint that `p1`'s value now equal the reference to this new point, and weak identity constraints that `p1` refer to what it used to (which can't be satisfied), and that `p2` refer to what it used to (which *can* be satisfied.)

The following program halts with an error, illustrating that we avoid clobbering the result of an assignment by making the value on right hand side of an assignment read-only.

```
p := Point.new(10,10);
always p.x = 100;
p := Point.new(50,50);
```

After the first statement, there is a point with `x=10,y=10` in the heap, and `p` refers to it. After the second statement, we add the constraint `p.x = 100` to the constraint store (closed over its environment of definition, so that `p` is bound appropriately). Then we solve the weak stay constraint on `p`'s identity, and then we evaluate the body of the constraint in constraint construction mode, yielding an integer constraint on the `x` field of `p`. This results in `p` continuing to refer to the same point, which now has 100 in its `x` field. So far so good. The third statement sets up an identity constraint between between `p` and the newly created point. However, since the right-hand side is annotated as read-only, and since this applies recursively to all parts and subparts, we can't change the `x` field of the new point to satisfy the `always p.x = 100` constraint, and instead halt with an error.

As with Babelsberg/UID, in the interests of predictability, the system will not create a new object to satisfy constraints. Thus, this program halts with an error:

```
p := Point.new(0,0);
q := p;
always p.x=5;
always q.x=10;
```

After the third statement, both `p` and `q` refer to the same point, which has `x=5,y=0`. Trying to satisfy the final constraint on `q.x` fails. We could satisfy the constraints by first cloning `q` from `p` and then changing its `x`, but we forbid creating new objects to satisfy constraints. In the semantics, this results in the `fail` state, just as would an unsatisfiable required constraint. (In a practical implementation, we might want to raise a different sort of exception in this case, to indicate that forcing an object to be created to satisfy a constraint is not allowed.)

While cloning a point seems innocuous, consider a similar example with windows on the user's display, which again results in an error. (It seems dangerous to silently create a second window.)

```
w1 := Window.new(....);
w2 := w1;
always w1.width = 200;
always w2.width = 300;
```

These variants, in which we make the constraints on the window's widths strong rather than required, or use a weak identity constraint instead of an assignment, result in an error as well. In these cases, there aren't any unsatisfied required constraints, but we would have weak constraints satisfied in preference to strong ones, which would violate the definition of correct solutions to the set of hard and soft constraints.

```
w1 := Window.new(....);
w2 := w1;
always strong w1.width = 200;
always strong w2.width = 300;
```

```
w1 := Window.new(....);
always weak w2 == w1;
always strong w1.width = 200;
always strong w2.width = 300;
```

As a final variant, this program is OK, since we aren't creating new objects to satisfy constraints and we don't have weak constraints being satisfied in preference to strong ones.

```
w1 := Window.new(....);
always strong w2 == w1;
always weak w1.width = 200;
always weak w2.width = 300;
```

As a point of comparison, this restriction on not creating new objects isn't relevant for Babelsberg/Records (although it is for Babelsberg/UID). Here is one of the above examples translated to Babelsberg/Records:

```
p := {x:0, y:0};
q := p;
always p.x=5;
always q.x=10;
```

This is fine — after the assignment `q := p`, `q` is a separate record from `p`, and so there is no problem satisfying the final constraint.

Here are a few examples to illustrate the behavior of weak identity constraints.

```
p := Point.new(0,0);
q := Point.new(5,5);
always p.x=10;
always q.x=20;
always weak p==q  /* this is a weak identity constraint */
```

This is OK; the final weak identity constraint isn't satisfied.

But consider the program with the constraints reordered:

```
p := Point.new(0,0);
q := Point.new(5,5);
always weak p==q;
always p.x=10;
always q.x=20;
```

After we satisfy the weak identity constraint, `p` and `q` will be aliased, and since we are disallowing the creation of new objects to satisfy constraints, we can't satisfy the last constraint, and the program halts with an error. It's a little weird that reordering the constraints has this effect, but particularly since in a practical system we would have a specific exception for the situation it should not present a program comprehension problem.

The same thing happens with `strong` rather than `required` constraints:

```
p := Point.new(0,0);
q := Point.new(5,5);
weak p==q;
always strong p.x=10;
always strong q.x=20;
```

7.4 Arrays

While it is not strictly necessary to include arrays in Babelsberg/Objects, we have used arrays in some examples that follow. We treat the length of an array is part of its structure, and use the same structural compatability checks for the existence of array elements that we do for the existence of fields in a record. Thus, we would not, for example, grow an array to allow a constraint on its i^{th} element to be satisfied if it didn't already have an i^{th} element.

7.5 Assignments in Methods Called by Constraints

Methods called by constraints can have assignment statements, just as any other method. Generally, however, this will mean that the method can only be used in the forwards direction (i.e., to compute the result given the inputs). Consider an iterative `sum` method for the class `Array`.

```
def sum
  ans := 0;
  i := 0;
  while i<self.length
    ans := ans + self[i];
    i := i+1;
  end;
  return ans;
end
```

This works as expected for normal imperative code, and can also be used in the forward direction in a constraint, for example:

```
a := Array.new(2);
a[0] := 10;
a[1] := 20;
always s = array.sum;
a[0] := 100;
```

After the `always` constraint is executed, `s` is 30; then after the final assignment to `a[0]`, `s` becomes 120.

However, if we attempted to constrain the sum of the array and expected that the system would update one or more elements to satisfy the constraint, the result would not be as desired — instead, the answer would be overridden to no effect. For example:

```
a := Array.new(2);
a[0] := 10;
a[1] := 20;
always 50 = array.sum;
```

Here `ans` in the `sum` method becomes 50 to satisfy the constraint, but since it was assigned to (which becomes a `once` constraint), rather than being permanently constrained, this result is lost. This is almost certainly not what the programmer intended, so in a practical language we should issue a warning. (The general class of warning here is that the result of a method call is being constrained or set, and that result doesn't have an `always` constraint on it, so will likely be lost.)

Generally, to write methods that work properly with constraints, the programmer should write the method using constraints rather than assignment statements. For example, here are two correct versions of the `sum` method. These use recursion, and will create as many partial sums as needed. With either of these methods, both of the above programs work as expected.

```
def sum
  if self.length=0
    then return 0
    else return self[0] + self[1..self.length].sum
  end;
end;

def sum
  return self.sum_from(0);
end;

def sum_from(start)
  if start >= self.length
    then return 0
    else return self[start] + self.sum_from(start+1);
  end;
end;
```

7.6 Formalism

This subsection will present the formal semantics for Babelsberg/Objects, including how constraints in the solver language are produced in constraint construction mode, and also all the usual object-oriented features such as inheritance, method lookup, and `self`. The key concepts here center around constraint construction mode and how it interacts with method calls; inheritance and method lookup is standard. So it might also be OK to reference an existing formalization, for example for Welterweight Java, and omit some of those parts.

7.7 Backwards Compatibility Mode for Methods

In Babelsberg/R, when methods are evaluated in constraint construction mode, assignment statements are converted to two-way equality constraints. This is useful for backward compatibility with existing code. (It won't allow an arbitrary method to be successfully used in all modes in a constraint, but it increases the chances that it will work.) However, this doesn't seem entirely clean, so we omit it from the formal semantics. Instead, a practical implementation of Babelsberg might elect to include a backwards compatibility mode for methods that does this transformation, as well as other transformations that increase the chances the method can be used in multi-directional constraints. In addition to converting assignments to constraints, the transformation should create fresh variables each time a variable is used on the left-hand-side of an assignment, and systematically use that new variable thereafter (until another assignment). First, here is an example that just uses straight-line code.

```
def add_and_double(x,y,z)
  sum := 0;
  sum := sum+x;
```

```
  sum := sum+y;
  sum := sum+z;
  return 2*sum;
end;
```

In backward compatibility mode this is rewritten as:

```
def add_and_double(x,y,z)
  sum_1 = 0;
  sum_2 = sum_1+x;
  sum_3 = sum_2+y;
  sum_4 = sum_3+z;
  return 2*sum_4;
end;
```

Notice that a fresh variable is introduced for each assignment, but not for the return statement. Now, if we evaluate the constraint `always 100 = add_and_double(10,15,n)`, the system should satisfy the constraint by making `n` be 25.

Conditionals, in which a variable might or might not be assigned to, are easily handled by using a required equality constraint to pass through the old value for the other branch of the conditional. For example:

```
def maybe_double(x)
  ans := x;
  if x<10
    then ans := 2*ans;
  end;
  return ans;
end;
```

This becomes:

```
def maybe_double(x)
  ans_1 = x;
  if x<10
    then ans_2 = 2*ans_1;
    else ans_2 = ans_1;
  end;
  return ans_2;
end;
```

Methods with loops could be converted automatically to recursions. Here is a version of `sum` that might be automatically generated in this fashion:

```
def sum
  helper(0,i,0,ans);
  return ans;
end;
```

```
def helper(old_i, new_i, old_ans, new_ans)
  if old_i<self.length
    then
      temp_ans = old_ans + self[old_i];
      temp_i = old_i + 1;
      helper(temp_i,new_i,temp_ans,new_ans);
    else
      new_ans = old_ans;
      new_i = old_i;
  end;
end;
```

Here, `helper` is produced automatically from the `while` by parameterizing the helper method with the old and new values of `i` and `ans`, and converting the `while` to an `if` with a recursive call.

7.8 Circular Structures

There is an issue regarding creating circular structures if we convert assignment statements to `once` constraints, with the right hand side annotated as read only. Consider:

```
c := Cons.new(10,nil);
c.cdr := c;
```

If the second assignment is converted to the constraint `once c.cdr == c?` this is unsatisfiable. However, there is a simple workaround, namely to replace such an assignment with a `once` identity constraint without the read-only annotation:

```
c := Cons.new(10,nil);
once c.cdr == c
```

This is a bit strange. But this solution doesn't require any changes to the formal semantics, and addresses all the previous issues. Also see Appendix A.2 for a practical variant that avoids this problem.

References

[1] Greg J Badros, Alan Borning, and Peter J Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[4] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. Babelsberg: Specifying and solving constraints on object behavior. Technical Report 81, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, May 2014. Also published as TR-2013-001, Viewpoints Research Institute, Los Angeles, CA.

[5] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/JS: A browser-based implementation of an object constraint language. In *Proceedings of the 2014 European Conference on Object-Oriented Programming*. Springer, July 2014. In press.

[6] Bjorn Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages, and Applications, and European Conference on Object-Oriented Programming*, pages 77–88, Ottawa, Canada, October 1990. ACM.

[7] Bjorn Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as Department of Computer Science and Engineering Technical Report 91-07-02.

[8] Bjorn Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[9] Bjorn Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268–286, June 1992.

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

[11] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, January 1987.

[12] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[13] Pierre Roy and François Pachet. Reifying constraint satisfaction in Smalltalk. *Journal of Object-Oriented Programming*, 10(4):43–51, 1997.

A Appendix

This appendix contains descriptions of some alternatives and observations that are potentially of interest, but that aren't essential to the main purpose of this note (which is to develop a formal semantics for Babelsberg).

A.1 Alternatives to Structural Compatability Checks

Our model includes structural compatibility checks on constraints, which allow us to desugar all of the record constraints (recursively) into constraints on the individual components. This means that the solver need not know anything about records. The alternative of having the solver know about records was rejected, since it is more complex and seems to lead to some odd cases, including requiring that the constraint solver spontaneously generate new kinds of records. For completeness, we outline that rejected alternative in this subsection.

We first consider records in Babelsberg/Records (immutable records, no UIDs).

If the solver does know about records, we need to extend the comparators to handle them. For LPB, there are at least two possibilities:

1. An equality constraint between records is either satisfied or it's not. For example consider a constraint `p=q` when `p={x:0, y:1}` and `q={x:0, y:2}`, versus when `q={a:1000}`. In both cases the constraint is unsatisfied, and there is no reason to prefer one of these solutions over the other.
2. An equality constraint between records is unfolded into multiple constraints on the fields, each of which is satisfied or not. (It might be unsatisfied either because the value in the field was wrong, or because the field doesn't even exist.) In the above example, for the first case we have `p.x=q.x` (satisfied) and `p.y=q.y` (unsatisfied). For the second case we have `p.x=q.x` (unsatisfied) and `p.y=q.y` (unsatisfied), and an additional constraint `p.a=q.a` (unsatisfied). The first solution would be preferred over the second under LPB.

We also considered WSB for records. The error for `{x:5}={x:5}` or `{x:5}={x:6}` is clear enough (0 and 1 respectively). But what is the error for `{x:5}={x:5,y:6}` or `{x:5}={y:6}`? We need to determine an error for additional or missing fields, and this error needs to be such that it can be meaningfully compared with the error for values of fields. Is the error for `{x:0}={x:1000000000}` less than for `{x:0}={y:0}`? We could come up with a definition, but it's not clear it's that useful.

Another problem with having the solver handle records is that it may result in less predictability for programmers. Consider:

```
p := {x:0}
q := {y:1}
always p=q
```

Here there are a number of plausible values for `p` and `q` — namely `{x:0}`, `{y:1}`, and `{x:0,y:1}` — and unclear which is better. The nondeterminism here is arguably worse than what arises with integers and floats and other values, since it affects the structure of the result rather than just its value.

Yet another complication is that we would want a minimality condition on records. Consider:

```
p.x = 5;
```

We want the solution to be `p={x:5}`, but not e.g. `p={x:5, y:1000}`.

These issues, along the lack of clear use cases for having a solver that can solve for records, led us to reject the alternative outlined in this subsection.

For Babelsberg/UID, an additional odd feature of having the solver know about records rather than using structural compatability checks is that the solver may need to spontaneously generate new records with their own UIDs. Consider:

```
a := {x:1};
b := a;
always a.x=1;
always b.x=2;
```

At first, `a` and `b` refer to the same record, but after the second `always` constraint, we need to create a new record and point `b` to it.

For Babelsberg/Objects, the analog of this behavior would be requiring that the constraint solver generate new classes on the fly; there will also be multiple possible classes. As before, this seems overly complex and unpredictable.

A.2 Alternatives for Creating Circular Structures

As discussed in Section 7.8, there is an issue with circular structures if we convert assignment statements to `once` constraints with the right hand side annotated as read only.

A.2.1 Selectively Eliding the Read-Only Annotation

A variant that provides a more standard syntax is to make the entire object on the right-hand side of the assignment be read-only, except for a field that is assigned to (if any). The constraints would be exactly the same for all the examples except for the circular structures one:

```
c := Cons.new(10,nil);
c.cdr := c;
```

Here, `c.car` and the reference to `c` itself are read-only, but not `c.cdr`.

This seems similar to the way that mutable state is modeled in Dedalus (where there is a persistence rule that says facts continue to hold, unless explicitly negated).

This variant seems mostly of practical interest — as far as the formal semantics is concerned, the simpler proposal described in Section 7.8 seems preferable.

A.2.2 Another Alternative: Distinguishing Ownership from Reference

Another alternative to making the entire object on the right-hand side of the assignment read-only is to distinguish ownership of parts from references to other objects. If an instance variable is for an "owned" part, then that instance variable is made read-only; but otherwise not. For compatibility with the host

language, the default would be that instance variables do not refer to owned parts; rather, this must be declared explicitly. (It only needs to be declared if there are constraints on the parts or subparts.)

This alternative does get rid of the workaround for creating circular structures, but is otherwise more complex, so we didn't select it.

A.3 Adding New Solvers and Extending the Solver Language

In a practical Babelsberg implementation, it is useful to be able to add new solvers (while still retaining the solver language), and also for simultaneously adding new solvers and extending the solver language so that new kinds of constraints can be encoded and sent to the new solvers. This doesn't seem to create any issues for the formal semantics, so we note this only in this appendix.

One useful extension of this sort would be adding a solver and constructs in the solver language for finite domain constraints, as in the clpfd library for SWI Prolog. Other possible solvers would be for strings, or for geometric constraints, either in 2 or 3 dimensions.

Another extension — already supported in Babelsberg/R and Babelsberg/JS — is to support local propagation constraints, as solved for example by DeltaBlue. Doing this requires adding a construct for declaring local propagation constraints (including the propagation methods) in the source language, and also adding local propagation constraints to the solver language.

In Babelsberg/JS, when a DeltaBlue constraint is constructed, and an equality constraint is encountered with two variables of the same type on either side, in constraint construction mode the interpreter doesn't descend further, but instead asserts the equality on those variables. So, for example, for two points the constraint expression does not desugar into multiple constraints on the x and y variables.

Finally, we may be able to have a solver language and solver to encode Prolog-like goals, which would support backtracking and Prolog-style programming within Babelsberg. A key observation here is that this would be in a separate set of constraints — we wouldn't try to change the basic Babelsberg semantics to allow backtracking with Babelsberg `if` statements.

A.4 Warnings and Debugging

A non-required `always` constraint might undo the result of an assignment.

```
x := 0;
always strong x=10;
x := 5;
```

`x` starts out as 0, then is 10 to satisfy the `always` constraint, then gets set to 5, and then at the next time step, the `always` constraint causes it to be set back to 10. Should we issue a warning in this case?