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

Tim Felgentreff, Todd Millstein, and Alan Borning

VPRI Technical Report TR-2014-002b
Draft of July 2015
(revised version of TR-2014-002, September 2014)

1 Introduction

Babelsberg [4] is a family of object constraint languages, with current instances being Babelsberg/R (a Ruby extension), Babelsberg/JS [5] (a Javascript extension), and Babelsberg/S (a Squeak extension). The Babelsberg design has evolved alongside its implementations, driven in part by practical considerations about the expectations of programmers familiar with the underlying host language. This fact, along with the complexities of integrating objects, state, and constraints, have led to a number of the semantic choices being muddled with implementation specifics. There have also been a number of long-standing, confusing issues with respect to constraints and object identity, how to represent assignment, and the appropriate restrictions on expressions that define constraints. In an effort to understand these better and to 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 variant we use in informal examples 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, inheritance, and object-oriented constraint definitions).

In each case, we first provide an informal discussion and examples, and then the formal semantics. The examples are designed to illustrate properties of our design in a concise manner. For all semantics given in this report, we provide an executable semantics that runs those examples that are also tests. These are highlighted in this report as follows:

Test 101

```
// code snippets with a margin tag like this one
// are part of the language spec test suite
x := 0;
y := x + 1
```

The primary audience for this admittedly long tech report is first ourselves — we have clarified many aspects of the language as a result of working out the formalism — and second, other researchers who are interested in the details. We are also writing a standard-length paper for a programming language conference that presents some of the key ideas from this tech report and that is intended to be of more general interest to the programming languages community.

2 Motivation

Our formal semantics is intended to provide a complete semantics of Babelsberg that can be used to inform practical implementations of the language. It is meant to be 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 focuses 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; instead we aim to make the smallest possible changes while still accommodating constraints in a clean and powerful way.

We require that our design support a useful and expressive language for constraints, integrated with the host object oriented language in a clean way. Beyond this, however, 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, in particular with respect to object identity and type). 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. Trying to follow this heuristic 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.

Some significant clarifications and simplifications of Babelsberg that have come out of this work on formalizing the language are:

- A clearer understanding of the interactions among constraints on identity, types, and values.
- The addition of structural compatibility checks to tame the power of the solver with respect to changing object structure and type to satisfy constraints.
- The addition of *value classes*. (Instances of value classes are immutable objects for which object identity is not significant.) Value classes play a key role in giving a clear specification of the requirements for expressions that define constraints (in particular that they must be side effect free), while still supporting useful programming idioms.
- A set of restrictions on identity constraints that make it easier to reason about object identity and type. In particular, any change to the identity of the object referred to by a variable flows clearly from a single assignment statement, and is deterministic — there are never multiple correct solutions to the

identity constraints. This also implies that any change to the type of a variable must similarly flow from a single assignment statement and be deterministic.

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 knowledge about how to handle hard and soft constraints is left entirely up to the solver and doesn't enter into the formal semantics, so the discussion in the rest of this section is intended to help the reader understand the informal examples — it isn't relevant to the formal semantics.

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 solver encodes the comparator being used, making it irrelevant to the formal semantics — however, in presenting the informal program examples we'll sometimes need to specify which is used, and for this reason we include a brief discussion of comparators.

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 [2] 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 three, namely "strong," "medium," 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. Read-only annotations provide an important tool in a practical language for guiding the behavior of the constraint solver. However, they don't present any particular issues for the formal semantics — we would just pass them on to the constraint solver to handle. Therefore, for simplicity we've dropped the extra rules to handle them (since they were just copies of the rules for variables without the read-only annotation). Finally, programs are often more concise if we permit read-only annotations on expressions as well as on variables. To accommodate this, the practical languages do a simple rewrite to convert a read-only expression to a read-only variable by introducing a fresh variable as needed — see Appendix A.2.

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 Taming Identity Constraints

We introduce object identity in the Babelsberg/UID language, and continue to use it in our final language Babelsberg/Objects. A central issue in the design of a constraint object language is the interaction between constraints and object identity. Our earlier experience with Kaleidoscope suggests that it is all too easy to make constraints on object identity and types be too powerful, so that they lead to non-obvious consequences. This makes programs more difficult for the programmer to understand. To tame the power of constraints on object identity and type, we set the following goals for this aspect of our design:

1. In those languages that include object identity, we want to support explicit but straightforward identity constraints of the form `x==y` (i.e., that variables `x` and `y` refer to the same object).
2. The solution to the constraints should be deterministic as far as object identity and type are concerned — there should never be multiple correct solutions in which a given variable refers to objects with different identities in the different solutions.
3. Any change to the identities of the objects referred to by variables should flow from an assignment statement — the constraint solver should not otherwise alter object identities.
4. To make Babelsberg programs more understandable for the programmer, we want to be able to reason about the constraints so that we can first solve all the identity constraints (with a deterministic solution, in keeping with Goal 2), and then the value constraints. A design that meets this goal of course benefits the language implementer as well.

In our intermediate language Babelsberg/Records, in which we introduce structured data in the form of immutable records but not yet object identity, we want analogous properties to hold: the solution to the constraints should be deterministic as far as object structure is concerned, and any changes to object structure should flow from an assignment statement.

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 find values for the variables in the constraint's expression such that it evaluates to true. So Boolean is a required datatype for all Babelsberg languages. In addition, we add reals as a second primitive type.

The evaluation model for Babelsberg/Reals is mostly standard for ordinary expressions and statements. In contrast, a statement that adds a constraint starts with a duration, namely `always` or `once`. The expression following the duration is taken unevaluated and added to the *constraint store*. For `always`, the constraint remains for the duration of the program's execution; for `once`, it is removed after the solver finds a solution to the current set of constraints. The only wrinkle in the evaluation model for ordinary statements is that for an assignment statement, we evaluate the expression on the right hand side of the assignment, constrain the variable on the left hand side of the assignment to be equal to the result using a `once` constraint, and then turn all the constraints over to the solver. Doing this ensures that assignment interacts correctly with other constraints.

Here are some examples.

Test 1

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

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 $\texttt{x} \mapsto \texttt{3}$.

Note that programs, as well as the variable names and values in the environment, are written in fixed pitch font. For contrast, we write the constraints that are handed to the solver in math font.

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 $\texttt{x} \mapsto \texttt{4}$.

The third statement adds the `always` constraint to the constraint store. 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 or least-squares-better, we get the solution $x = 10$, since this minimizes the error for the weak constraint. If we use locally-predicate-better, then every $x \in [10, \infty)$ is a solution, and the system is free to select any of them. (However, as noted in Section 3, typically we wouldn't use LPB if we have inequalities.)

The following example illustrates using the same variable on the left and right hand sides of an assignment statement, as well as the interaction of assignments with `always` constraints.

Test 2

```
x := 3;
y := 0;
always y = x+100;
x := x+2
```

After evaluating the first two statements and solving the resulting constraints, the environment has the binding $\texttt{x} \mapsto \texttt{3}$, $\texttt{y} \mapsto 0$. The third 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 $x = 3$
weak $y = 0$
required $y = x + 100$

This has multiple possible solutions, and the solver can select any one of them. Suppose it picks $\texttt{x} \mapsto 3$, $\texttt{y} \mapsto 103$.

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

weak $x = 3$
weak $y = 103$
required $y = x + 100$
required $x = 5$

The first two constraints are the weak stays on `x` and `y`, the third comes from the always constraint in the constraint store, and the fourth 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.

New variables must be created with an assignment statement. Thus the following program is illegal — we would need to create `x` first with an assignment statement before adding the `always` constraint.

Test 3
```
always x=10;
```

Requiring that variables be created before equating them with something can be annoying for the programmer. We decided that indeed the above program is illegal in the formal semantics. However, in practical implementations, we can have a shortcut to allow it. It only works for = constraints where one side is a new variable and all variables on the other side already exist. The behavior is that the system creates the new variable, evaluates the other side of the equality, assigns it to the new variable, and then adds the equality constraint as an `always` constraint.

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, as was done in for example the first Babelsberg paper [4]. See Appendix A.4 for a discussion of these alternatives. (This is relegated to the appendix since, while interesting, it is largely orthogonal to the task at hand of providing a formal semantics for Babelsberg).

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

Test 4
```
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:

Test 5
```
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.

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 be free of side effects.[1]
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 — 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):

Test 6

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

For simplicity, our formal rules don't include short-circuit evaluation — adding it would be straightforward but would require additional, not-very-interesting rules.

Constraints with conjunctions or disjunctions are just turned over to the solver, rather than being evaluated using short-circuit evaluation. We could also add `if` expressions to the language (distinct from `if` statements). However, since there is a simple translation from `if` expressions to conjunctions and disjunctions, we don't include them. (If we did have them, they would also need to simply be turned over to the solver.) For example, the following two constraints are equivalent, and have the solution `x` $\mapsto$ 10:

```
x := 0;
always if x=4 then x=5 else x=10
```

Test 7

```
x := 0;
always (x=4 and x=5) or (x!=4 and x=10)
```

In either case, we would turn the entire constraint over to the solver to find a solution for `x`.

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

[1] In a practical implementation, the programmer might be able to make cautious use of benign side effects in a constraint expression, for example, for caching or constructing temporary objects that are garbage collected before they are visible outside the constraint. In the formal semantics, however, we simply disallow side effects in constraint expressions.

```
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 values (such as records, arrays, or sets).

Since we are modeling a dynamically typed language, we need to consider the case of changing the type of a variable. Here's a simple example program that illustrates this.

Test 8

```
x := 5;
x := "Hello";
```

After the second statement we solve the following constraints:

weak	$x = 5$
required	$x =$ `"Hello"`

The final result is `x` $\mapsto$ `"Hello"`.

The changed type might propagate through an `always` constraint:

Test 9

```
x := 5;
y := 10;
always y = x;
x := "Hello";
```

The final result is that both `x` and `y` are strings.

There is also a potential interaction with overloaded operators. Suppose that for strings `+` denotes string concatenation:

Test 10

```
x := 5;
y := 10;
always y = x+x;
x := "Hello";
```

After the `always` statement we have `x` $\mapsto$ `5`, `y` $\mapsto$ `10`. Then after the final statement, we have the constraints

weak	$x = 5$
weak	$y = 10$
required	$y = x + x$
required	$x =$ `"Hello"`

This has the solution `x` $\mapsto$ `"Hello"`, `y` $\mapsto$ `"HelloHello"`.

Non-determinism can arise for primitive types as well as values:

Test 11

```
x := 3;
always weak x=5;
always weak x="hello";
```

Here the solver is free to choose either an integer or a string for `x`. This is strange, and also not really in keeping with the spirit of the goals for our languages with respect to constraints and object identity (Section 3.2). However, it is just an artifact of the semantic rules we are using for Babelsberg/PrimitiveTypes — for simplicity, in these rules we don't distinguish among primitive types, with the consequence that the above program is legal. When we get to our final goal (the Babelsberg/Objects language) the corresponding program will be disallowed due to the automatic boxing and unboxing of primitive types in the formal semantic rules.

4.4 Formalism

We present the formal semantics of Babelsberg/PrimitiveTypes.

4.4.1 Syntax

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

The language includes a set of boolean and base type constants (e.g., reals), ranged over by metavariable `c`. A finite set of operators on expressions is ranged over by $\oplus$. This includes operations on the reals such as `+` and `*`, a set of *predicate* operators ($=$ and $\neq$, $\leq$, $<$, $=$, and so on. It also includes a set of logical operators for combining boolean expressions (e.g., $\wedge$, $\vee$). The predicate operations are assumed to include at least an equality operator `=` for each primitive type in the language, and the logical operations 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 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 a top element `required`. While syntax requires the priority to be explicit, for simplicity we sometimes omit it in this semantics. A constraint with no explicit priority implicitly has the priority `required`. Finally, for simplicity we do not model read-only annotations in the formal semantics.

The syntax is thus that of a simple, standard imperative language except for the `always` and `once` statements, which declare constraints. An `always` constraint must hold for the rest of the programs execution, whereas a `once` constraint is satisfied by the solver and then retracted. Note that for simplicity this semantics implicitly gets stuck whenever the solver cannot satisfy a constraint, either due to an unsatisfiable constraint or due to the solver being unable to determine whether the constraint is satisfiable. In a practical implementation, we would likely want to differentiate between these cases, 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.

4.4.2 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. When convenient we also view an environment as a set of (program variable, program value) pairs. For each operator $\oplus$ in the language we assume the existence of a corresponding semantic function denoted $[\![\oplus]\!]$.

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

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

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

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

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

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

This judgment represents a call to the constraint solver, which we treat as a black box. The proposition $\texttt{E} \models \texttt{C}$ denotes that environment `E` is a *solution* to the constraint `C` (and further one that is optimal according to the solver's semantics, as discussed earlier).

$\boxed{\text{stay}(\texttt{x=v}, \rho) = \texttt{C}}$

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

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

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

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

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

"Execution starting from configuration <E|C|s> ends in state <E′|C′>."

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. Note that our execution implicitly gets stuck if the solver cannot produce a model.

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{v} \qquad \text{stay}(\texttt{E}, \rho) = \texttt{C}_s \qquad \texttt{E}' \models (\texttt{C} \wedge \texttt{C}_s \wedge \texttt{x = v})}{\texttt{<E|C|x := e>} \longrightarrow \texttt{<E}'\texttt{|C>}} \quad \text{(S-Asgn)}$$

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

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

$$\texttt{<E|C|skip>} \longrightarrow \texttt{<E|C>} \quad \text{(S-Skip)}$$

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

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

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

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

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

5 Babelsberg/Records

For this next language, we augment Babelsberg/PrimitiveTypes with immutable records. This language is interesting on its own as an expository language and as a step toward Babelsberg/Objects; but we will also continue to use these immutable records in the formal semantics for Babelsberg/Objects to represent *value classes* (see Section 7).

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

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

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:

Test 12

```
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 q = p;       /* an equality constraint between two records */
always q.y = 20;
```

Note the difference between the assignment `q:=p` and the constraint `always q=p`. After the `always p.x=100` constraint, we have `p` $\mapsto$ {`x:100, y:5`} but `q` $\mapsto$ {`x:2, y:5`} — at this point `p` and `q` are unrelated, so adding the constraint on `p.x` had no effect on `q`. However, after the final constraint, we have `p` $\mapsto$ {`x:100, y:20`} and `q` $\mapsto$ {`x:100, y:20`}, since `p` and `q` are now constrained to be equal.

The solver must now also handle records. To tame the power of the solver so that it does not (for example) invent new fields for records, we add *structural compatibility checks* on constraints. These structural compatibility checks are assertions that are checked dynamically before sending the constraints involving records to the solver, for example, checking whether a variable is bound to a record, and whether the record has the necessary fields. While these assertions are checked, unlike 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.

Here are a few examples of structural compatibility checks.

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

The structural compatibility check is that `p` is a record that has an `x` field, which succeeds.

The following program is OK — just as in Babelsberg/PrimitiveTypes, we can change the type of a variable using an assignment.

Test 13

```
a := {x:1};
a := {y:10};
```

However, in contrast to Babelsberg/PrimitiveTypes, the following program fails the structural compatibility checks — only an assignment can change the type of a variable.

Test 14

```
a := {x:1};
once a = {y:10};
```

This program fails the structural compatibility checks as well:

Test 15

```
a := {x:1};
b := {x:1};
always a=b;
a := {x:1, y:10};
```

Here, assigning a record with a different structure to `a` is OK on its own, but the `always` constraint would also require `b` to change. It's a bit weird that this behavior is different from the behavior with primitive types (in which a similar program was OK). However, once we introduce object identity, we will be able to have changes to the types of variables ripple through the system via identity constraints (but it will need to be via identity constraints and not value constraints).

A few more examples:

These two programs fail, because as in Babelsberg/PrimitiveTypes only assignment can initialize variables.

```
once a = {y:10};
```

Test 16

```
a := {y:10};
always b=a;
```

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

Test 17

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

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

Test 18

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

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

Test 19

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

However, the following program is OK:

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

The solution will be that `p` is a record with a single field `x` bound to a number, with the exact value depending on the solver (3, 4, or 3.5 being the most reasonable possibilities). Note that the solver may need to adjudicate among solutions that have different primitive types in a given field, but the structural compatibility checks ensure that it doesn't need to decide between two records with e.g. different numbers of fields.

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

Test 20

```
a := {x:0};
b := {y:5};
always a=b
```

This is the case even though there actually are records that would satisfy the required `a=b` constraint — the issue is that the assignment to `a` leaves it with a record type that has a single `x` field, the assignment to `b` leaves it with a record type that has a single `y` field, and the structural compatibility check prevents the solver from changing either of these types. This illustrates one aspect of using the structural compatibility checks to tame the solver — one could otherwise imagine the solver coming up with the solution `a` $\mapsto$ {`x:0, y:5`} and `b` $\mapsto$ {`x:0, y:5`}, or even `a` $\mapsto$ {`x:0, y:5, z:10`} and `b` $\mapsto$ {`x:0, y:5, z:10`}. See Appendix A.6 for more about this issue.

5.1 Formalism

5.1.1 Syntax

The syntax from Section 4 is augmented now to support records and the ability to access fields of a record:

Expression	$\texttt{e}$	$::=$	$\cdots \mid \{\texttt{l}_1\texttt{:e}_1,\ldots,\texttt{l}_n\texttt{:e}_n\} \mid \texttt{e.l}$
Label	$\texttt{l}$	$::=$	record label names
Value	$\texttt{v}$	$::=$	$\texttt{c} \mid \{\texttt{l}_1\texttt{:v}_1,\ldots,\texttt{l}_n\texttt{:v}_n\}$

We now assume the solver "understands" records and record operations directly. We assume the equality operator `=` can be used to compare two records for logical equality, but no other operators apply to record values.

5.1.2 Semantics

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

Expression evaluation is updated to support records.

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

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

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

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

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

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

We use a notion of typechecking to prevent undesirable non-determinism in constraints. Specifically, we want constraint solving to preserve the structure of the values of variables, changing only the underlying primitive data as part of a solution, in support of the goals listed in Section 3.2. We formalize our notion of structure through a simple syntax of types:

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

The typechecking rules are mostly standard. However, we check expressions dynamically just before constraint solving, so we typecheck in the context of a runtime environment, which is somewhat unusual.

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

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

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

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

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

The rule T-Field ensures that a field of a record can only be referenced when it already exists in the record. This is necessary to ensure that the solver will never have to invent record fields in order to satisfy a constraint containing field accesses.

The rule T-Op ensures that two records can only be combined with binary operations (e.g. to compare them) when they have the same structure. This is necessary to ensure that the solver will never have to invent record fields in order to satisfy a constraint that combines two records through equality or other.

The following two rules simply ensure that a constraint is well typed.

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

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

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

The solving judgment is unchanged from Babelsberg/PrimitiveTypes.

$\boxed{\text{stay}(\mathtt{x}{=}\mathtt{v},\ \rho) = \mathtt{C}}$

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

We add a stay rule to express stays on records. For records, we create fresh variables for each field and add a stay on the variable to refer to a record with exactly the same fields, and the fresh variables as values. We then add stay constraints for each of those variables to be equal to the current value of the field. When solving in S-Once, the stay on the structure will be required, so the structure of records cannot change when adding a fresh constraint. During assignment, however, the stay will be weak, so that the structure *can* change through imperative updates.

$$\frac{\mathtt{x}_1 \text{ fresh} \cdots \mathtt{x}_n \text{ fresh} \qquad \text{stay}(\mathtt{x}_1{=}\mathtt{v}_1,\ \rho) = \mathtt{C}_1 \cdots \text{stay}(\mathtt{x}_n{=}\mathtt{v}_n,\ \rho) = \mathtt{C}_n}{\text{stay}(\mathtt{x} = \{\mathtt{l}_1{:}\mathtt{v}_1, \ldots, \mathtt{l}_n{:}\mathtt{v}_n\},\ \rho) = (\rho\ \mathtt{x} = \{\mathtt{l}_1{:}\mathtt{x}_1, \ldots, \mathtt{l}_n{:}\mathtt{x}_n\}) \ \wedge \ \mathtt{C}_1 \ \wedge \cdots \wedge \ \mathtt{C}_n} \quad \text{(StayRecord)}$$

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

The semantics for executing statements is essentially identical to what we had before, except that now we typecheck constraints before we solve them. Implicitly we get "stuck" if such a check fails. In a practical language, this could be extended to generate an exception instead. We only show the modified rules below.

$$\frac{\texttt{E} \vdash \texttt{e} \Downarrow \texttt{v} \qquad \texttt{E[x} \mapsto \texttt{v]} \vdash \texttt{C} \qquad \text{stay}(\texttt{E}, \texttt{weak}) = \texttt{C}_s \qquad \texttt{E}' \models (\texttt{C} \wedge \texttt{C}_s \wedge \texttt{x = v})}{\texttt{<E|C|x := e>} \longrightarrow \texttt{<E}'\texttt{|C>}} \text{ (S-Asgn)}$$

The second premise above is necessary so that we don't pass the solver an ill-typed set of constraints. Specifically, since the assignment can change the type of `x`, we have to make sure that the global constraints in scope, `C`, are well typed with respect to this new type. Note that the stay constraint for `x` within $\texttt{C}_s$ may not be well typed with respect to the new type of `x`, but that's OK since the required constraint `x = v` will take precedence. The syntax `E[x` $\mapsto$ `v]` denotes the environment identical to `E` but with `x` mapped to the value `v`.

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

5.2 Adding Mutable Records

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

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

After the second assignment, we would have `p` $\mapsto$ `{x:100, y:0}`. This doesn't add any particular complications to the semantics.

We can equivalently express this program without mutable records as follows:

Test 21

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

However, 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 by making a new record whose fields are simply copied from those in the old record, except for the field being assigned. For example, the above program is equivalent to the following program in Babelsberg/Records:

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

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}`.

Mutable records would thus simply provide a syntactic convenience, rather than some new capability. We would still want immutable records as well (which we will use to model instances of value classes in the formal semantics for Babelsberg/Objects), so this would also add clutter to the languages. We therefore don't include Babelsberg with mutable records 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 with object identity on the heap, we use the keyword `new` when creating one. The syntax includes field accesses as both r-values and l-values.

In all the examples in this section, we will only use records with object identity. However, later, when we are formalizing Babelsberg/Objects, we will use both kinds of records: records with object identity to model objects that have identity, and immutable records (as in Babelsberg/Records) to model instances of value classes (which are immutable objects for which object identity is not significant). It should be clear from context when we use the word "record" what kind we mean; but when it's necessary to distinguish them, we'll use the terms "i-record" and "uid-record" for immutable records and records with object identity respectively.

Only uid-records live on the heap — primitive values and i-records do not. (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 or i-record on the heap, it is easy to simulate this by constructing a uid-record that has a single field that holds the integer or boolean or i-record.)

A field of a uid-record holds either a primitive type, a i-record, or a reference to another uid-record. There is no syntax for creating a nested uid-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 uid-records could be supported directly with just a source code transformation. In any case, when we get to Babelsberg/Objects, we will remove this restriction and allow expressions that do involve creating new objects.)

Since object identity is now significant, we add identity constraints to the language (following Smalltalk syntax, written `==`, in contrast to `=` for equality constraints). 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 and Babelsberg/Records, the weak stays on variables are identity constraints rather than equality constraints. For uid-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. In addition, there are also weak stay constraints on the values in the record fields.

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

Here are some examples involving uid-record fields and constraints:

Test 22

```
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. As before, our model includes structural compatibility checks on constraints, which tames the solver so that it doesn't do such things as inventing new records or adding fields to a record. (The only way to create a new record is with an explicit `new` expression in the program, and this also defines all of the fields that it has and will ever have.)

After executing the right hand side of the first statement, the heap is updated with a new object and a reference r to point to it. The assignment and stay constraints then desugar into:

required	$p = r$
required	$\mathtt{H}(r) = \{x : x_r, y : y_r\}$
weak	$x_r = 2$
weak	$y_r = 5$

The reference r is a constant. We assume the solver understands uninterpreted function symbols. The uninterpreted function symbol H in the second constraint above is used to represent the heap's mapping from references to their contents; the constraint forces the solver to keep the reference r pointing to a record that has *only* x and y fields. We introduce variables for the values of these fields and add weak stays so these keep the values with which they were created — these can always be satisfied, because the variables are fresh.

After the second statement we have the stay constraints on `p` and its fields, and the required constraint resulting from the assignment:

weak	$p = r$
required	$\mathtt{H}(r) = \{x : x_r, y : y_r\}$
weak	$x_r = 2$
weak	$y_r = 5$
required	$a = 2$

After the third statement:

weak	$p = r$
required	$\mathtt{H}(r) = \{x : x_r, y : y_r\}$
weak	$x_r = 2$
weak	$y_r = 5$
weak	$a = 2$
required	$p.x = 6$

And finally:

weak	$p = r$
required	$\mathtt{H}(r) = \{x : x_r, y : y_r\}$
weak	$x_r = 6$
weak	$y_r = 5$
weak	$a = 2$
required	$p.x = 100$

After solving the final set of constraints we have the environment $\mathtt{p} \mapsto \mathtt{r} \wedge \mathtt{a} \mapsto \mathtt{2}$, as well as the heap $\mathtt{r} \mapsto$ `{x:100, y:5}`.

Here is an example that fails the structural compatibility checks.

Test 23

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

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

The following example demonstrates one form of aliasing:

Test 24

```
p := new {x:2, y:5};
q := p;
p.x := 100;
q := new {z:10};
p.x := 200;
```

After the assignment `q:=p`, `q` holds the same reference as `p`, so the subsequent assignment to `p.x` changes both. But then we break the alias with another assignment to `q`, so the second assignment to `p.x` doesn't affect `q` (which by that point no longer has an `x` field).

There can also be explicit identity constraints:

Test 25

```
p := new {x:2, y:5};
q := p;
always q==p;
q := new {z:10};
```

After the last statement both `p` and `q` point to the same `{z:10}` record. Note that we don't have structural compatibility checks for re-satisfying identity constraints, so that the assignment to `q` that changes its structure ripples through the identity constraint to change `p` as well. However, if the `always` constraint had been a record equality constraint rather than an identity constraint, the last assignment to `q` would have failed the structural compatibility check, which would have expected it to be a record with `x` and `y` fields.

In Section 3.2 we listed a number of goals for our design to attempt to tame the power of identity constraints and the solver. In support of this, we require that variables be created in an assignment statement — the above program would have been illegal if we didn't have the `q:=p` statement but just tried to create `q` with the `always` constraint. We also require that a new identity constraint be satisfied at the time it is created — otherwise it's an error (modeled in the formal semantics by the rules getting stuck). This restriction was obeyed in the above program. Here is an example that violates it, and is hence not legal:

Test 26

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

Without the restriction, after the last statement `p` and `q` would definitely point to the same record, but it might be either `{x:2}` or `{y:5}`. (In any case, `p` and `q` would still one of the records that was already created by the `new` expressions — the solver wouldn't invent a new one — but we want to tame the language further.)

With the restriction, any changes to the types of variables must flow from an explicit assignment statement — the effects can still ripple outwards via identity constraints — but they are deterministic. For the same reason, we disallow disjunctions of identity constraints (since otherwise there could be programs with multiple correct solutions to the identity constraints). It is only updates to values that can be non-deterministic as the result of constraints. One item for future work will be to formalize and prove this statement, which seems like a useful property for the language.

As another aspect of taming identity constraints, explicit priorities on identity constraints are not allowed — there are only the implicit weak identity stay constraints on all variables. (We haven't found any compelling

use cases for programmer-specified soft identity constraints, and omitting them simplifies reasoning about identity constraints for the programmer.) For simplicity, there are no explicit conjunctions of identity constraints, only the implicit conjunction resulting from writing several of them.

This program is thus illegal, since it uses explicit priorities on identity constraints:

Test 27
```
p := new {x:0};
q := new {x:5};
always medium p.x = 0;
always medium q.x = 5;
always weak p==q
```

Without the restriction, the programmer would need to interleave determining values and identities in reasoning about the program's behavior. For example, for the above program, to decide that we should leave the weak identity constraint unsatisfied, we'd need to first solve the value constraints and then decide that we can't satisfy the weak identity constraint.

The solver can't spontaneously create new uid-records if they are needed — new uid-records can only be created with an explicit assignment with a `new` on the right hand side. Consider:

Test 28
```
a := new {x:1};
b := a;
always a.x=1;
always b.x=2;
```

After `b:=a`, `a` and `b` refer to the same record, but after the second `always` constraint, the solver would need to spontaneously create a new record and point `b` at it to satisfy the constraints. So this program halts with an unsatisfiable constraint error.

Similarly, the solver is not allowed to create a new record when an L-value is used on the left-hand side of an assignment. So the next program fails, and the solver does not spontaneously create a record `x` with a field `l`:

Test 29
```
x.l := 10
```

As a third example of this kind, the next program is also illegal – the last line makes the constraint unsatisfiable. The solver could satisfy is by adding a field `b` to the new object assigned to `x`, but we also disallow this. Thus, when assigning `x` on the last line, the preceding value constraint no longer typechecks.

Test 30
```
x := new {b:0}
y := new {a:x}
always y.a.b == 0
x := new {c:0}
```

The solver also can't switch identities around to satisfy value constraints. Here is the a program, where a record with the required value for its `x` field happens to be lying around. But to no avail: this program halts with an unsatisfiable constraint error as well, rather than silently changing `b` to refer to the `{x:2}` record. Our formal semantics for Babelsberg/UID does not enforce that, but the full semantics for Babelsberg/Objects does, by including appropriate rules for translating constraints to the solver. For now, we just implicitly assume that there is a mechanism that prevents the solver from changing `b`.

Test 31
```
a := new {x:1};
b := a;
c := new {x:2};
always a.x=1;
```

```
always b.x=2;
```

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. Note that in this version we temporarily remove records as values. This is to make the proofs for our theorems more concise. We will re-add them in the full Babelsberg/Objects language.

6.1.1 Syntax

We use the following syntax for Babelsberg/UID:

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

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

In the syntax, we treat H as a keyword used for dereferencing. Source programs will not use expressions of the form H(e), but they are introduced as part of constraints given to the solver, which we assume will treat H as an uninterpreted function. We also assume that the solver supports records and record equality, which we also denote with the = operator.

6.1.2 Operational Semantics

The semantics includes an environment E and a heap H. The former is a function that maps variable names to values, while the latter is a function that maps mutable references to "objects" of the form $\{l_1:v_1,\ldots,l_n:v_n\}$. When convenient, we also treat both E and H as a set of pairs ({(x,v),...} and {(r,o),...}, respectively). The currently active value constraints are kept as a compound constraint C; identity constraints are kept as a single conjunction referred to as I.

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

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

The rules for evaluation are mostly as expected in an imperative language. We do not give rules for expressions of the form `H(e)`, because they are not meant to appear in source. For each operator $\oplus$ in the language we assume the existence of a corresponding semantic function denoted $[\![\oplus]\!]$.

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

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

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

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

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

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

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

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

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

"Expression `e` 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 a notion of typechecking to prevent undesirable non-determinism in constraints. Specifically, we want constraint solving to preserve the structure of the values of variables, changing only the underlying primitive data as part of a solution. We formalize our notion of structure through a simple syntax of types:

Type `T` ::= `PrimitiveType` | $\{\texttt{l}_1\texttt{:T}_1\texttt{,...,l}_n\texttt{:T}_n\}$

The typechecking rules are mostly standard. We check expressions dynamically just before constraint solving, so we typecheck in the context of a runtime environment. Note that we do not include type rules for identities. This ensures that constraints involving them do not typecheck, so identity checks cannot occur in ordinary constraints.

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

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

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

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

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

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

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

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

This judgment represents a call to the constraint solver, which we treat as a black box. The proposition $\mathtt{E;H} \models \mathtt{C}$ denotes that environment `E` and heap `H` are an *optimal solution* to the constraint `C`, according to the solver's semantics.

We assume several well-formedness properties about a solution `E` and `H` to constraints `C`:

- any object reference appearing in the range of `E` also appears in the domain of `H`
- any object reference appearing in the range of `H` also appears in the domain of `H`
- for all variables `x` in the domain of `E` there is some type `T` such that $\mathtt{E;H} \vdash \mathtt{x} : \mathtt{T}$
- $\mathtt{E;H} \vdash \mathtt{C}$

$\boxed{\text{stay}(\mathtt{x=v},\ \rho) = \mathtt{C}}$

$\boxed{\text{stay}(\mathtt{r=o},\ \rho) = \mathtt{C}}$

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

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

As in Kaleidoscope, the semantics ensure that each variable has a stay constraint to keep it at its current value, if possible. The stay rules take a priority as a parameter. When solving value constraints, this priority is set to `required`, to ensure that the structures of objects and the relationship between l-values and object

references cannot change. When solving identity constraints as part of executing an assignment statement, the priority is set to `weak` to allow structural changes.

To properly account for the heap in the constraint solver, we employ an uninterpreted function `H` that maps references to objects (i.e., records). The rules below employ this function in order to define stay constraints for references.

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

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

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

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

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

$\boxed{\text{stayPrefix}(\texttt{E},\, \texttt{H},\, \texttt{L}) = \texttt{C}}$

$\boxed{\text{stayPrefix}(\texttt{E},\, \texttt{H},\, \texttt{I}) = \texttt{C}}$

These judgments are another form of stay constraints that ensure that the "prefix" `L` of an l-value `L.l` is unchanged; this is necessary to ensure that updates to the value of `L.l` are deterministic.

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

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

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

$\boxed{\texttt{E;H} \vdash \texttt{C} \rightsquigarrow \texttt{C}'}$

We use these judgments to translate a constraint into a constraint suitable for the solver. Specifically, each occurrence of an expression of the form `L.l`, where `L` refers to a heap reference `r`, is translated into `H(L).l` (recursively, as required), and each occurrence of the identity operator `==` is replaced by ordinary equality. We do not give these rules, because they are straightforward.

$\boxed{\text{solve}(\texttt{E},\, \texttt{H},\, \texttt{C},\, \rho) = \texttt{E}'\texttt{;H}'}$

"Solving constraint C in the context of E and H using stay constraints with priority ρ produces the new environment and heap E′ and H′."

This judgment represents one phase of constraint solving – either solving "value" constraints or identity constraints.

$$\frac{\text{stay}(\mathtt{E}, \rho) = \mathtt{C}_E \qquad \text{stay}(\mathtt{H}, \rho) = \mathtt{C}_H \qquad \mathtt{E};\mathtt{H} \vdash \mathtt{C} \rightsquigarrow \mathtt{C}' \qquad \mathtt{E}';\mathtt{H}' \models (\mathtt{C}' \ \wedge \ \mathtt{C}_E \ \wedge \ \mathtt{C}_H)}{\text{solve}(\mathtt{E}, \mathtt{H}, \mathtt{C}, \rho) = \mathtt{E}';\mathtt{H}'} \quad \text{(SOLVE)}$$

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

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

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

The rule below describes the semantics of assignments. We employ a two-phase process. First the identity constraints are solved in the context of the new assignment. This phase propagates the effect of the assignment through the identities, potentially changing the structures of objects as well as the relationships among objects in the environment and heap. In the second phase, the value constraints are typechecked against the new environment and heap resulting from the first phase. If they are well typed, then we proceed to solve them. This phase can change the values of primitives but will not modify the structure of any object.

Implicitly this rule gets stuck if either a) the identity constraints cannot be solved, b) the value constraints do not typecheck, or c) the value constraints cannot be solved. A practical implementation would add explicit exceptions for these cases that the programmer could handle.

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

The next rule describes the semantics of object creation, which is straightforward. For simplicity we require a new object to be initially assigned to a fresh variable, but this is no loss of expressiveness.

$$\frac{\mathtt{E}(\mathtt{x})\uparrow \qquad \mathtt{H}(\mathtt{r})\uparrow \qquad \mathtt{E}' = \mathtt{E} \bigcup \{(\mathtt{x}, \mathtt{r})\} \qquad \mathtt{H}' = \mathtt{H} \bigcup \{(\mathtt{r}, \{\mathtt{l}_1{:}\mathtt{v}_1, \ldots, \mathtt{l}_n{:}\mathtt{v}_n\})\} \qquad \mathtt{E};\mathtt{H} \vdash \mathtt{e}_1 \Downarrow \mathtt{v}_n \cdots \mathtt{E};\mathtt{H} \vdash \mathtt{e}_n \Downarrow \mathtt{v}_n}{\texttt{<E|H|C|I|x := new } \{\mathtt{l}_1{:}\mathtt{e}_1, \ldots, \mathtt{l}_n{:}\mathtt{e}_n\}\texttt{>} \longrightarrow \texttt{<E′|H′|C|I>}} \quad \text{(S-ASGNNEW)}$$

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

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

$$\frac{\texttt{<E|H|C|I|once } L_0 \texttt{ == } L_1\texttt{>} \longrightarrow \texttt{<E|H|C|I>} \qquad I' = I \wedge L_0 == L_1}{\texttt{<E|H|C|I|always } L_0 \texttt{ == } L_1\texttt{>} \longrightarrow \texttt{<E|H|C|I'>}} \text{ (S-AlwaysIdentity)}$$

The following two rules describe the semantics of value constraints. Recall that these constraints cannot contain identity constraints in them (because identity constraints do not typecheck). As we show later, solving value constraints cannot change the structure of any objects on the environment and heap.

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

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

The remaining rules are standard for imperative languages, only augmented with constraint stores, and are only given for completeness.

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

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

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

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

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

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

6.1.3 Properties

Here we state and prove two key theorems about our formalism.

We assume that a given configuration `E|H|C|I` is *well formed*, meaning that it satisfies these sanity conditions:

- any object reference appearing in the range of `E` also appears in the domain of `H`
- any object reference appearing in the range of `H` also appears in the domain of `H`
- for all variables `x` in the domain of `E` there is some type `T` such that $\texttt{E;H} \vdash \texttt{x} : \texttt{T}$
- $\texttt{E;H} \vdash \texttt{C}$
- $\texttt{E;H} \vdash \texttt{I} \Downarrow \texttt{true}$

Well-formedness follows from the assumptions made on the constraint solver described earlier.

The first theorem formalizes the idea that any solution to a value constraint preserves the structures of the objects on the environment and heap:

Theorem 1 *(Structure Preservation) If* $\texttt{<E|H|C|I|s>} \longrightarrow \texttt{<E'|H'|C'|I'>}$ *and* $\texttt{s}$ *either has the form* $\texttt{once C}_0$ *or* $\texttt{always C}_0$ *and* $\texttt{E;H} \vdash \texttt{C}_0$ *and* $\texttt{E;H} \vdash \texttt{L} : \texttt{T}$*, then* $\texttt{E';H'} \vdash \texttt{L} : \texttt{T}$.

Proof. If $\texttt{s}$ has the form $\texttt{once C}_0$ then the result follows by Lemma 1. If $\texttt{s}$ has the form $\texttt{always C}_0$, then since $\texttt{<E|H|C|I|s>} \longrightarrow \texttt{<E'|H'|C'|I'>}$ and $\texttt{C}_0$ is not an identity test, by rule S-Always we have $\texttt{<E|H|C|I|once C}_0\texttt{>} \longrightarrow \texttt{<E'|H'|C|I>}$. Then again the result follows from Lemma 1. □

Lemma 1 *If* $\texttt{<E|H|C|I|once C}_0\texttt{>} \longrightarrow \texttt{<E'|H'|C'|I'>}$ *and* $\texttt{E;H} \vdash \texttt{C}_0$ *and* $\texttt{E;H} \vdash \texttt{L} : \texttt{T}$*, then* $\texttt{E';H'} \vdash \texttt{L} : \texttt{T}$.

Proof. Since $\texttt{<E|H|C|I|once C}_0\texttt{>} \longrightarrow \texttt{<E'|H'|C'|I'>}$ and $\texttt{C}_0$ is not an identity test, by S-Once we have that $\texttt{E;H} \vdash \texttt{C}_0$ and $\text{solve}(\texttt{E}, \texttt{H}, \texttt{C} \wedge \texttt{C}_0, \texttt{required}) = \texttt{E';H'}$. By the assumption of well-formedness we have $\texttt{E;H} \vdash \texttt{C}$ so by T-Conjunction also $\texttt{E;H} \vdash \texttt{C} \wedge \texttt{C}_0$. Therefore the result follows from Lemma 2. □

Lemma 2 *If* $\texttt{E;H} \vdash \texttt{C}$ *and* $\text{solve}(\texttt{E}, \texttt{H}, \texttt{C}, \texttt{required}) = \texttt{E';H'}$ *and* $\texttt{E;H} \vdash \texttt{L} : \texttt{T}$*, then* $\texttt{E';H'} \vdash \texttt{L} : \texttt{T}$.

Proof. Since $\text{solve}(\texttt{E}, \texttt{H}, \texttt{C}, \texttt{required}) = \texttt{E';H'}$ by Solve we have that $\text{stay}(\texttt{E}, \texttt{required}) = \texttt{C}_{E_s}$ and $\text{stay}(\texttt{H}, \texttt{required}) = \texttt{C}_{H_s}$ and $\texttt{E;H} \vdash \texttt{C} \rightsquigarrow \texttt{C'}$ and $\texttt{E';H'} \models (\texttt{C'} \wedge \texttt{C}_{E_s} \wedge \texttt{C}_{H_s})$. We prove this lemma by structural induction on $\texttt{T}$.

By Lemma 3 there exists a value $\texttt{v}$ such that $\texttt{E;H} \vdash \texttt{L} \Downarrow \texttt{v}$ and $\texttt{E;H} \vdash \texttt{v} : \texttt{T}$. Then by Lemma 4 there exists a value $\texttt{v'}$ such that $\texttt{E';H'} \vdash \texttt{L} \Downarrow \texttt{v'}$. Furthermore, if $\texttt{v}$ is an object reference $\texttt{r}$, then also $\texttt{v'} = \texttt{r}$. Case analysis on the form of $\texttt{v'}$:

- Case $\texttt{v'}$ is a constant $\texttt{c'}$. Then also $\texttt{v}$ is a constant $\texttt{c}$. Since $\texttt{E;H} \vdash \texttt{v} : \texttt{T}$, the last rule in this derivation must be T-Const so $\texttt{T}$ is $\texttt{PrimitiveType}$ and the result follows from T-Const.
- Case $\texttt{v'}$ is a reference $\texttt{r'}$. Since $\texttt{E';H'} \vdash \texttt{L} \Downarrow \texttt{v'}$ by Lemma 5 there is a type $\texttt{T'}$ such that $\texttt{E';H'} \vdash \texttt{L} : \texttt{T'}$ and $\texttt{E';H'} \vdash \texttt{r'} : \texttt{T'}$. So it suffices to show that $\texttt{T} = \texttt{T'}$. Case analysis on the form of $\texttt{v}$:
  - Case $\texttt{v}$ is a constant $\texttt{c}$. Since $\texttt{E;H} \vdash \texttt{v} : \texttt{T}$, the last rule in this derivation must be T-Const, so we have $\texttt{T} = \texttt{PrimitiveType}$. Case analysis on the last rule in the derivation of $\texttt{E;H} \vdash \texttt{L} \Downarrow \texttt{v}$:
    - Case E-Variable. Then $\texttt{L}$ is a variable $\texttt{x}$ and $\texttt{E}(\texttt{x}) = \texttt{c}$. Since $\text{stay}(\texttt{E}, \texttt{required}) = \texttt{C}_{E_s}$, by StayEnv we have that $\texttt{C}_{E_s}$ includes a conjunct $\texttt{C}_x$ such that $\text{stay}(\texttt{x=c}, \texttt{required}) = \texttt{C}_x$, and by StayConst $\texttt{C}_x$ has the form $\texttt{weak x=c}$.
      Since $\texttt{E';H'} \vdash \texttt{L} \Downarrow \texttt{r'}$, the last rule in this derivation must be E-Variable, so we have $\texttt{E'}(\texttt{x}) = \texttt{r'}$. We know that assigning $\texttt{x}$ to $\texttt{c}$ satisfies the weak stay constraint above. Therefore there must be some constraint on $\texttt{x}$ in $\texttt{C}$ that causes the change in value for $\texttt{x}$ from $\texttt{c}$ to $\texttt{r'}$.
    - Case E-Field. Then $\texttt{L}$ has the form $\texttt{L}_0\texttt{.l}_i$ and $\texttt{E;H} \vdash \texttt{L}_0 \Downarrow \texttt{r}_0$ and $\texttt{H}(\texttt{r}_0) = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}$ and $1 \leq i \leq n$ and $\texttt{r'} = \texttt{v}_i$. Since $\text{stay}(\texttt{H}, \texttt{required}) = \texttt{C}_{H_s}$, by StayHeap we have $\texttt{C}_{H_s}$ includes a conjunct $\texttt{C}_{r_0}$ such that $\text{stay}(\texttt{r}_0\texttt{=}\{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}, \texttt{required}) = \texttt{C}_{r_0}$. By StayObject $\texttt{C}_{r_0}$ has the form $\texttt{(required H(r}_0\texttt{)=}\{\texttt{l}_1\texttt{:x}_1\texttt{,...,l}_n\texttt{:x}_n\}\texttt{)} \wedge \texttt{C}_1 \wedge \cdots \wedge \texttt{C}_n$, where $\texttt{x}_1 \ldots \texttt{x}_n$ are fresh variables and $\text{stay}(\texttt{x}_i\texttt{=v}_i, \texttt{required}) = \texttt{C}_i$. Then by StayConst $\texttt{C}_i$ has the form $\texttt{weak x}_i\texttt{=c}$.

Since $\mathtt{E;H} \vdash \mathtt{L}_0 \Downarrow \mathtt{r}_0$, by Lemma 4 we have $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L}_0 \Downarrow \mathtt{r}_0$. Since $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L} \Downarrow \mathtt{r'}$, the last rule in this derivation must be E-Field, so we have that $\mathtt{H'}(\mathtt{r}_0) = \{\mathtt{l}_1:\mathtt{v}'_1,\ldots,\mathtt{l}_n:\mathtt{v}'_n\}$ where $\mathtt{v}'_i$, and hence $\mathtt{x}_i$ is $\mathtt{r'}$. We know that assigning $\mathtt{x}_i$ to $\mathtt{c}$ satisfies the weak stay constraint above. Therefore there must be some constraint on an l-value of the form $\mathtt{L}_{00}.\mathtt{l}_i$ in $\mathtt{C}$, where $\mathtt{E;H} \vdash \mathtt{L}_{00} \Downarrow \mathtt{r}_0$, that causes the change in value for $\mathtt{x}_i$ from $\mathtt{c}$ to $\mathtt{r'}$.

Therefore in either case, there must be some constraint on an l-value $\mathtt{L''}$ in $\mathtt{C}$, where $\mathtt{E;H} \vdash \mathtt{L''} \Downarrow \mathtt{c}$ but $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L''} \Downarrow \mathtt{r'}$. By Lemma 5 there is some $\mathtt{T''}$ such that $\mathtt{E;H} \vdash \mathtt{L''} : \mathtt{T''}$ and $\mathtt{E;H} \vdash \mathtt{c} : \mathtt{T''}$. The last rule in the derivation of $\mathtt{E;H} \vdash \mathtt{c} : \mathtt{T''}$ must be T-Const so $\mathtt{T''}$ is `PrimitiveType`. We argue a contradiction by case analysis of the immediate parent expression of any occurrence of $\mathtt{L''}$ in $\mathtt{C}$. We are given $\mathtt{E;H} \vdash \mathtt{C}$ so this parent expression must also be well typed.

- Case $\mathtt{L''}.\mathtt{l}$. Then by T-Field $\mathtt{L''}$ must have a record type, contradicting the fact that $\mathtt{E;H} \vdash \mathtt{L''} :$ `PrimitiveType`.
- Case $\mathtt{L''}$ is an immediate subexpression of an $\oplus$ operation. But these operations only apply to primitives, so the solver cannot satisfy them by assigning $\mathtt{x}$ to $\mathtt{r'}$.
- Case $\rho\ \mathtt{L''}$. This constraint is only satisfied if $\mathtt{x}$ is a boolean, so the solver will not assign $\mathtt{x}$ to $\mathtt{r'}$.

- Case $\mathtt{v}$ is a reference $\mathtt{r}$. Then $\mathtt{r} = \mathtt{r'}$. Since $\mathtt{E;H} \vdash \mathtt{v} : \mathtt{T}$, by T-Ref we have that $\mathtt{H}(\mathtt{r})=\{\mathtt{l}_1:\mathtt{v}_1,\ldots,\mathtt{l}_n:\mathtt{v}_n\}$ and $\mathtt{E;H} \vdash \mathtt{v}_1 : \mathtt{T}_1 \cdots \mathtt{E;H} \vdash \mathtt{v}_n : \mathtt{T}_n$ and $\mathtt{T}$ is $\{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}$. By T-Field we have $\mathtt{E;H} \vdash \mathtt{L}.\mathtt{l}_i : \mathtt{T}_i$ for each $1 \le i \le n$, so by induction $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L}.\mathtt{l}_i : \mathtt{T}_i$ for each $1 \le i \le n$.

  Since stay(`H`, `required`) $= \mathtt{C}_{H_s}$, by StayHeap we have that $\mathtt{C}_{H_s}$ includes a conjunct $\mathtt{C}_r$ such that stay($\mathtt{r}=\{\mathtt{l}_1:\mathtt{v}_1,\ldots,\mathtt{l}_n:\mathtt{v}_n\}$, `required`) $= \mathtt{C}_r$. By StayObject $\mathtt{C}_r$ has the form (`required` $\mathtt{H(r)}=\{\mathtt{l}_1:\mathtt{x}_1,\ldots,\mathtt{l}_n:\mathtt{x}_n\}$) $\wedge\ \mathtt{C}_1 \wedge \cdots \wedge \mathtt{C}_n$, where $\mathtt{x}_1 \ldots \mathtt{x}_n$ are fresh variables and stay($\mathtt{x}_i=\mathtt{v}_i$, `required`) $= \mathtt{C}_i$. Therefore any solution to $(\mathtt{C'} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s})$ must map $\mathtt{r}$ to an object value of the form $\{\mathtt{l}_1:\mathtt{v}'_1,\ldots,\mathtt{l}_n:\mathtt{v}'_n\}$ in $\mathtt{H'}$.

  Since $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L}.\mathtt{l}_i : \mathtt{T}_i$ for each $1 \le i \le n$, by Lemma 3 we have $\mathtt{E;H} \vdash \mathtt{L}.\mathtt{l}_i \Downarrow \mathtt{v}''_i$ and $\mathtt{E;H} \vdash \mathtt{v}''_i : \mathtt{T}_i$ for each $1 \le i \le n$. Since the last rule in each of these evaluation derivations must be E-Field we have that $\mathtt{v}''_i = \mathtt{v}'_i$ for each $1 \le i \le n$. Finally, since $\mathtt{E'};\mathtt{H'} \vdash \mathtt{r} : \mathtt{T'}$, the last rule in this derivation must be T-Ref, so we have that $\mathtt{T'} = \{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}$.

□

Lemma 3 *If* $\mathtt{E;H} \vdash \mathtt{L} : \mathtt{T}$*, then there exists a value* $\mathtt{v}$ *such that* $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{v}$ *and* $\mathtt{E;H} \vdash \mathtt{v} : \mathtt{T}$.

Proof. By structural induction on $\mathtt{L}$:

- Case $\mathtt{L}$ is a variable $\mathtt{x}$: Then by T-Var we have that $\mathtt{E}(\mathtt{x}) = \mathtt{v}$ and $\mathtt{E;H} \vdash \mathtt{v} : \mathtt{T}$. Finally by E-Var we have $\mathtt{E;H} \vdash \mathtt{x} \Downarrow \mathtt{v}$.
- Case $\mathtt{L}$ has the form $\mathtt{L'}.\mathtt{l}_i$: By T-Field we have that $\mathtt{E;H} \vdash \mathtt{L'} : \{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}$ and $1 \le i \le n$ and $\mathtt{T} = \mathtt{T}_i$. By induction there exists a value $\mathtt{v'}$ such that $\mathtt{E;H} \vdash \mathtt{L'} \Downarrow \mathtt{v'}$ and $\mathtt{E;H} \vdash \mathtt{v'} : \{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}$. Case analysis of the derivation of $\mathtt{E;H} \vdash \mathtt{v'} : \{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\}$:
  - Case T-Const: Then $\{\mathtt{l}_1:\mathtt{T}_1,\ldots,\mathtt{l}_n:\mathtt{T}_n\} =$ `PrimitiveType` and we have a contradiction.
  - Case T-Ref: Then we have $\mathtt{v'} = \mathtt{r}$ and $\mathtt{H}(\mathtt{r})=\{\mathtt{l}_1:\mathtt{v}_1,\ldots,\mathtt{l}_n:\mathtt{v}_n\}$ and $\mathtt{E;H} \vdash \mathtt{v}_i : \mathtt{T}_i$. Finally, by E-Field we have $\mathtt{E;H} \vdash \mathtt{L'}.\mathtt{l}_i \Downarrow \mathtt{v}_i$.

□

Lemma 4 *If* $\mathtt{E;H} \vdash \mathtt{C}$ *and* solve($\mathtt{E}$, $\mathtt{H}$, $\mathtt{C}$, `required`) = $\mathtt{E'};\mathtt{H'}$ *and* $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{v}$*, then there exists a value* $\mathtt{v'}$ *such that* $\mathtt{E'};\mathtt{H'} \vdash \mathtt{L} \Downarrow \mathtt{v'}$*. Furthermore, if* $\mathtt{v}$ *is an object reference* $\mathtt{r}$*, then also* $\mathtt{v'} = \mathtt{r}$*.*

Proof. Since solve($\mathtt{E}$, $\mathtt{H}$, $\mathtt{C}$, `required`) = $\mathtt{E'};\mathtt{H'}$ by SOLVE we have that stay($\mathtt{E}$, `required`) = $\mathtt{C}_{E_s}$ and stay($\mathtt{H}$, `required`) = $\mathtt{C}_{H_s}$ and $\mathtt{E;H} \vdash \mathtt{C} \rightsquigarrow \mathtt{C'}$ and $\mathtt{E';H'} \models (\mathtt{C'} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s})$. We proceed by structural induction on $\mathtt{L}$:

- Case $\mathtt{L}$ is a variable $\mathtt{x}$. Since $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{v}$, by E-VAR we have that $\mathtt{E}(\mathtt{x}) = \mathtt{v}$. Then since stay($\mathtt{E}$, `required`) = $\mathtt{C}_{E_s}$, by STAYENV we have that $\mathtt{C}_{E_s}$ includes a conjunct $\mathtt{C}_x$ such that stay(`x=v`, `required`) = $\mathtt{C}_x$. Case analysis of the rule used in the derivation of stay(`x=v`, `required`) = $\mathtt{C}_x$:
  - STAYCONST: Then $\mathtt{C}_x$ has the form `weak x=v` and $\mathtt{v}$ is a constant $\mathtt{c}$. Therefore the variable $\mathtt{x}$ appears in the constraint $(\mathtt{C'} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s})$, so any solution to the constraint must include some value $\mathtt{v'}$ for $\mathtt{x}$ in $\mathtt{E'}$, and the result follows by E-VAR.
  - STAYREF: Then $\mathtt{C}_x$ has the form `required x=v` and $\mathtt{v}$ is an object reference $\mathtt{r}$. Therefore any solution to $(\mathtt{C'} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s})$ must map $\mathtt{x}$ to $\mathtt{r}$ in $\mathtt{E'}$, and the result follows by E-VAR.
- Case $\mathtt{L}$ has the form $\mathtt{L'.l}_i$. Then by E-FIELD we have $\mathtt{E;H} \vdash \mathtt{L'} \Downarrow \mathtt{r}$ and $\mathtt{H}(\mathtt{r}) = \{\mathtt{l}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v}_n\}$ and $1 \leq i \leq n$ and $\mathtt{v} = \mathtt{v}_i$. By induction we have $\mathtt{E';H'} \vdash \mathtt{L'} \Downarrow \mathtt{r}$. Since stay($\mathtt{H}$, `required`) = $\mathtt{C}_{H_s}$, by STAYHEAP we have that $\mathtt{C}_{H_s}$ includes a conjunct $\mathtt{C}_r$ such that stay($\mathtt{r}=\{\mathtt{l}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v}_n\}$, `required`) = $\mathtt{C}_r$. By STAYOBJECT $\mathtt{C}_r$ has the form $\mathtt{(required\ H(r)=\{l_1:x_1,\ldots,l_n:x_n\})} \wedge \mathtt{C}_1 \wedge \cdots \wedge \mathtt{C}_n$, where $\mathtt{x}_1 \ldots \mathtt{x}_n$ are fresh variables and stay($\mathtt{x}_i\mathtt{=v}_i$, `required`) = $\mathtt{C}_i$. Therefore any solution to $(\mathtt{C'} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s})$ must map $\mathtt{r}$ to an object value of the form $\{\mathtt{l}_1\mathtt{:v'}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v'}_n\}$ in $\mathtt{H'}$, where $\mathtt{v'}_i$ is the value assigned to variable $\mathtt{x}_i$ by the solution. Therefore by E-FIELD we have $\mathtt{E;H} \vdash \mathtt{L'.l}_i \Downarrow \mathtt{v'}_i$. Finally suppose $\mathtt{v}_i$ is an object reference $\mathtt{r}_i$. Then by STAYREF $\mathtt{C}_i$ has the form $\mathtt{required\ x}_i\mathtt{\ =\ r}_i$ so any solution to the constraints must map $\mathtt{x}_i$ to $\mathtt{r}_i$, so also $\mathtt{v'}_i$ is $\mathtt{r}_i$.

□

Lemma 5 *If* $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{v}$ *then there is some type* $\mathtt{T}$ *such that* $\mathtt{E;H} \vdash \mathtt{L} : \mathtt{T}$ *and* $\mathtt{E;H} \vdash \mathtt{v} : \mathtt{T}$*.*

Proof. By induction on the derivation of $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{v}$:

- Case E-VARIABLE. Then $\mathtt{L}$ is a variable $\mathtt{x}$ and $\mathtt{E}(\mathtt{x}) = \mathtt{v}$. Then by well formedness there is some type $\mathtt{T}$ such that $\mathtt{E;H} \vdash \mathtt{x} : \mathtt{T}$ and by `T-Var` also $\mathtt{E;H} \vdash \mathtt{v} : \mathtt{T}$.
- Case E-FIELD. Then $\mathtt{L}$ has the form $\mathtt{L'.l}_i$ and $\mathtt{E;H} \vdash \mathtt{L'} \Downarrow \mathtt{r}$ and $\mathtt{H}(\mathtt{r}) = \{\mathtt{l}_1\mathtt{:v}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:v}_n\}$ and $1 \leq i \leq n$ and $\mathtt{v} = \mathtt{v}_i$. By induction there is some type $\mathtt{T'}$ such that $\mathtt{E;H} \vdash \mathtt{L'} : \mathtt{T'}$ and $\mathtt{E;H} \vdash \mathtt{r} : \mathtt{T'}$. Then by T-REF $\mathtt{T'}$ has the form $\{\mathtt{l}_1\mathtt{:T}_1\mathtt{,}\ldots\mathtt{,l}_n\mathtt{:T}_n\}$ where $\mathtt{E;H} \vdash \mathtt{v}_i : \mathtt{T}_i$. Then by T-FIELD also $\mathtt{E;H} \vdash \mathtt{L} : \mathtt{T}_i$.

□

Lemma 6 *If* $\mathtt{E;H} \vdash \mathtt{L} \Downarrow \mathtt{r}$ *then* $\mathtt{r}$ *is in the domain of* $\mathtt{H}$*.*

Proof. Case analysis on the last rule in the evaluation derivation.

- Case E-VARIABLE. Then $\mathtt{L}$ is a variable $\mathtt{x}$ and $\mathtt{E}(\mathtt{x}) = \mathtt{r}$. Then the result follows from the assumption that $\mathtt{E}$ and $\mathtt{H}$ are well formed.

- Case E-Field. Then L has the form $\texttt{L}'\texttt{.l}_i$ and $\texttt{E;H} \vdash \texttt{L}' \Downarrow \texttt{r}'$ and $\texttt{H}(\texttt{r}) = \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}$ and $1 \leq i \leq n$ and $\texttt{r} = \texttt{v}_i$. Then the result follows from the assumption that E and H are well formed.

□

The second theorem formalizes the idea that all solutions to an assignment will produce structurally equivalent environments and heaps:

Theorem 2 *(Structural Determinism) If* $\texttt{<E|H|C|I|L := e>} \longrightarrow \texttt{<E}_1\texttt{|H}_1\texttt{|C}_1\texttt{|I}_1\texttt{>}$ *and* $\texttt{<E|H|C|I|L := e>} \longrightarrow \texttt{<E}_2\texttt{|H}_2\texttt{|C}_2\texttt{|I}_2\texttt{>}$ *and* $\texttt{E;H} \vdash \texttt{x} : \texttt{T}_0$*, then there exists a type* T *such that* $\texttt{E}_1\texttt{;H}_1 \vdash \texttt{x} : \texttt{T}$ *and* $\texttt{E}_2\texttt{;H}_2 \vdash \texttt{x} : \texttt{T}$*.*

Proof. By S-Asgn we have $\texttt{E;H} \vdash \texttt{e} \Downarrow \texttt{v}$ and stayPrefix(E, H, L) = $\texttt{C}_L$ and stayPrefix(E, H, I) = $\texttt{C}_I$ and solve(E, H, $\texttt{C}_L \wedge \texttt{C}_I \wedge \texttt{L=v} \wedge \texttt{I}$, weak) = $\texttt{E}'_1\texttt{;H}'_1$ and $\texttt{E}'_1\texttt{;H}'_1 \vdash \texttt{C}$ and solve($\texttt{E}'_1$, $\texttt{H}'_1$, $\texttt{C} \wedge \texttt{L=v}$, required) = $\texttt{E}_1\texttt{;H}_1$. Also by S-Asgn and Lemma 13 we have solve(E, H, $\texttt{C}_L \wedge \texttt{C}_I \wedge \texttt{L=v} \wedge \texttt{I}$, weak) = $\texttt{E}'_2\texttt{;H}'_2$ and $\texttt{E}'_2\texttt{;H}'_2 \vdash \texttt{C}$ and solve($\texttt{E}'_2$, $\texttt{H}'_2$, $\texttt{C} \wedge \texttt{L=v}$, required) = $\texttt{E}_2\texttt{;H}_2$. By Lemma 7 we have that $\texttt{E}'_1 = \texttt{E}'_2$ and $\texttt{H}'_1 = \texttt{H}'_2$. Since $\texttt{E;H} \vdash \texttt{x} : \texttt{T}_0$, by T-Var x is in the domain of E, so it is also in the domain of $\texttt{E}'_1$ and by E-Var we have $\texttt{E}'_1\texttt{;H}'_1 \vdash \texttt{x} \Downarrow \texttt{v}_x$ where $\texttt{E}'_1(\texttt{x}) = \texttt{v}_x$. Then by Lemma 5 there is some type $\texttt{T}'$ such that $\texttt{E}'_1\texttt{;H}'_1 \vdash \texttt{v}_x : \texttt{T}'$ and $\texttt{E}'_1\texttt{;H}'_1 \vdash \texttt{x} : \texttt{T}'$. Finally by Lemma 2 we have that $\texttt{E}_1\texttt{;H}_1 \vdash \texttt{x} : \texttt{T}'$ and $\texttt{E}_2\texttt{;H}_2 \vdash \texttt{x} : \texttt{T}'$, so the result is shown with $\texttt{T} = \texttt{T}'$.

□

Definition 1 *We say that* $\texttt{L}_1$ *and* $\texttt{L}_2$ *are* aliases *given* E *and* H *if either:*

- $\texttt{L}_1$ *and* $\texttt{L}_2$ *are the same variable* x
- $\texttt{L}_1$ *has the form* $\texttt{L}'_1\texttt{.l}$ *and* $\texttt{L}_2$ *has the form* $\texttt{L}'_2\texttt{.l}$ *and there is a reference* r *such that* $\texttt{E;H} \vdash \texttt{L}'_1 \Downarrow \texttt{r}$ *and* $\texttt{E;H} \vdash \texttt{L}'_2 \Downarrow \texttt{r}$

Definition 2 *We say that* L *and* $\texttt{L}'$ *are the* operands *of the constraint* $\texttt{L == L}'$*.*

Definition 3 *We define the* induced graph *of* I *and* L *given* E *and* H *as follows. Let S be the set that includes* L *as well as all operands of identity tests in* I*. Partition S into equivalence classes defined by the alias relation:* $\texttt{L}_1$ *and* $\texttt{L}_2$ *are in the same equivalence class if and only if they are aliases given* E *and* H*. Then the induced graph has one node per equivalence class and an undirected edge between nodes* N_1 *and* N_2 *if there is a conjunct* $\texttt{L}_1\texttt{==L}_2$ *in* I *such that* $\texttt{L}_1$ *belongs to node* N_1 *and* $\texttt{L}_2$ *belongs to node* N_2*.*

Definition 4 *We say that a node in the induced graph of* I *and* L *given* E *and* H *is* relevant *if it is reachable from* L*'s node in the graph; an l-value is* relevant *if its node in the graph is relevant.*

Definition 5 *The* relevant update *of* E *and* H *for* I *and* L=v *is the environment* $\texttt{E}'$ *and heap* $\texttt{H}'$ *that are identical to* E *and* H *except that for each relevant l-value* $\texttt{L}_0$ *in the induced graph of* I *and* L *given* E *and* H*:*

- *If* $\texttt{L}_0$ *is a variable* x*, then* $\texttt{E}'(\texttt{x}) = \texttt{v}$*.*
- *If* $\texttt{L}_0$ *has the form* $\texttt{L}'_0\texttt{.l}$ *and* $\texttt{E;H} \vdash \texttt{L}'_0 \Downarrow \texttt{r}$*, then* $\texttt{E}'\texttt{;H}' \vdash \texttt{r.l} \Downarrow \texttt{v}$*.*

Lemma 7 *If* stayPrefix(E, H, L) = $\texttt{C}_L$ *and* stayPrefix(E, H, I) = $\texttt{C}_I$ *and* solve(E, H, $\texttt{C}_L \wedge \texttt{C}_I \wedge \texttt{L=v} \wedge \texttt{I}$, weak) = $\texttt{E}_1\texttt{;H}_1$ *and* solve(E, H, $\texttt{C}_L \wedge \texttt{C}_I \wedge \texttt{L=v} \wedge \texttt{I}$, weak) = $\texttt{E}_2\texttt{;H}_2$*, then* $\texttt{E}_1 = \texttt{E}_2$ *and* $\texttt{H}_1 = \texttt{H}_2$*.*

Proof. By SOLVE we have stay(E, weak) = C_E and stay(H, weak) = C_H and E;H $\vdash$ ($C_L \wedge C_I \wedge$ L=v $\wedge$ I) $\rightsquigarrow$ C' and E′;H′ $\models$ ($C' \wedge C_E \wedge C_H$) and E″;H″ $\models$ ($C' \wedge C_E \wedge C_H$).

By Lemma 9, E′ and H′ include all the updates of the relevant update (which we will call E_0 and H_0) of E and H for I and L=v, and similarly for E″ and H″. To complete the proof we argue that both of these solutions are in fact identical to the relevant update. WLOG we consider E′ and H′. By Lemma 8 the relevant update of E and H for I and L=v is a solution to the constraint $C' \wedge C_E \wedge C_H$. Therefore if E′ and H′ is not the relevant update, then by Definition 5 either:

- there is a variable x such that E_0(x) = E(x) but E′(x) has a different value
- there is a reference r and field label l such that H_0(r).l = H(r).l but H′(r).l has a different value

Consider the former. Since stay(E, weak) = C_E, by STAYENV, STAYCONST, and STAYREF we have that C_E includes a weak constraint x=v_x, where E(x) = v_x. Since E′ and H′ include all the updates of the relevant update, E′ and H′ satisfy strictly fewer weak constraints than the relevant update, contradicting the optimality of E′ and H′.

Similarly, consider the latter. Since stay(H, weak) = C_H, by STAYHEAP and STAYOBJECT C_H includes a required constraint H(r)={l_1:x_1,...,l_n:x_n} where the x_i variables are fresh and l is some l_i. Then by STAYCONST and STAYREF there is a weak constraint of the form x_i = v_i where H(r).l_i = v_i. Since E′ and H′ include all the updates of the relevant update, E′ and H′ satisfy strictly fewer weak constraints than the relevant update, contradicting the optimality of E′ and H′.

□

Lemma 8 *If* stay(E, weak) = C_E *and* stay(H, weak) = C_H *and* stayPrefix(E, H, L) = C_L *and* stayPrefix(E, H, I) = C_I *and* E;H $\vdash$ ($C_L \wedge C_I \wedge$ L=v $\wedge$ I) $\rightsquigarrow$ C' *and the constraint* $C' \wedge C_E \wedge C_H$ *is satisfiable, then the relevant update* E′ *and* H′ *of* E *and* H *for* I *and* L=v *is a solution to the constraint* $C' \wedge C_E \wedge C_H$.

Proof. It suffices to show that all required constraints in $C' \wedge C_E \wedge C_H$ are satisfied in E′ and H′. We consider the various constraints in turn:

- C_E: Since stay(E, weak) = C_E, by STAYENV, STAYCONST, and STAYREF there are no required constraints in C_E, so all required constraints are satisfied vacuously.
- C_H: Since stay(H, weak) = C_H, by STAYHEAP, STAYOBJECT, STAYCONST, and STAYREF the only required constraints in C_H have the form H(r)={l_1:x_1,...,l_n:x_n} where the x_i variables are fresh and H maps r to some value of the form {l_1:v_1,...,l_n:v_n}. By Definition 5 also H′ maps r to a value of the form {l_1:v'_1,...,l_n:v'_n} so the constraint is satisfied.
- C_L: By STAYPREFIXFIELD the conjuncts in C_L have the form x=v or r.l=v. Suppose the relevant update fails to satisfy one of these conjuncts. We consider each form in turn:
  - x=v: Then the relevant update maps x to some v′ ≠ v in E′. But by SOLVE and Lemma 9 any solution to the constraint $C' \wedge C_E \wedge C_H$ must map x to v′ in the environment, which violates the constraint x=v. So there must be no solution to the constraints, contradicting our assumption of satisfiability.
  - r.l=v: Then the relevant update maps r.l to some v′ ≠ v in H′. But by SOLVE and Lemma 9 any solution to the constraint $C' \wedge C_E \wedge C_H$ must map r.l to v′ in the environment, which violates the constraint r.l=v. So there must be no solution to the constraints, contradicting our assumption of satisfiability.

- $\mathtt{C}_I$: By StayPrefixIdent the conjuncts in $\mathtt{C}_L$ have the form `x=v` or `r.l=v`. So the argument is identical to that above for the $\mathtt{C}_L$ constraint.
- `L=v`: By Definitions 3 and 5 we have that `L` is relevant for `I` and `L` given `E` and `H`. Then by Lemma 10 we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L} \Downarrow \mathtt{v}$, so by E-Op and the semantics of equality we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L=v} \Downarrow \mathtt{true}$.
- `I`: Consider a conjunct $\mathtt{L}_0$ `==` $\mathtt{L}_1$ in `I`. We have two cases:
  - $\mathtt{L}_0$ is relevant for `I` and `L` given `E` and `H`. By Definitions 4 and 3, $\mathtt{L}_1$ is also relevant. Then by Lemma 10 we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_1 \Downarrow \mathtt{v}$, so by E-IdentityTrue we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0\mathtt{==}\mathtt{L}_1 \Downarrow \mathtt{true}$.
  - $\mathtt{L}_0$ is not relevant for `I` and `L` given `E` and `H`. By Definitions 4 and 3, $\mathtt{L}_1$ is also not relevant. By well formedness we have $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0\mathtt{==}\mathtt{L}_1 \Downarrow \mathtt{true}$, so by E-IdentityTrue there is some $\mathtt{v}_0$ such that $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$ and $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_1 \Downarrow \mathtt{v}_0$. Then by Lemma 11 we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_1 \Downarrow \mathtt{v}_0$, so by E-IdentityTrue we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0\mathtt{==}\mathtt{L}_1 \Downarrow \mathtt{true}$.

□

Lemma 9 *If* stayPrefix(`E`, `H`, `L`) = $\mathtt{C}_L$ *and* stayPrefix(`E`, `H`, `I`) = $\mathtt{C}_I$ *and* solve(`E`, `H`, $\mathtt{C}_L \wedge \mathtt{C}_I \wedge$ `L=v` $\wedge$ `I`, `weak`) = $\mathtt{E}';\mathtt{H}'$ *and* $\mathtt{L}_0$ *is a relevant l-value of* `I` *and* `L` *given* `E` *and* `H`*, then* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}$ *and:*

- *If* $\mathtt{L}_0$ *is a variable* `x`*, then* $\mathtt{E}'(\mathtt{x}) = \mathtt{v}$.
- *If* $\mathtt{L}_0$ *has the form* $\mathtt{L}_0'$`.l` *and* $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0' \Downarrow \mathtt{r}$*, then* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{r.l} \Downarrow \mathtt{v}$.

Proof.

By Definition 4, we know that $\mathtt{L}_0$'s node in the induced graph for `I` and `L` given `E` and `H` is reachable from `L`'s node. The proof proceeds by induction on the length k of the path between these nodes.

- Case $k = 0$. Then by Definition 3, `L` and $\mathtt{L}_0$ are aliases given `E` and `H`. Case analysis on the structure of `L`:
  - Case `L` is a variable `x`. Then by Definition 1 also $\mathtt{L}_0$ is `x`. Since solve(`E`, `H`, $\mathtt{C}_L \wedge \mathtt{C}_I \wedge$ `L=v` $\wedge$ `I`, `weak`) = $\mathtt{E}';\mathtt{H}'$ we must have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{x=v} \Downarrow \mathtt{true}$, so by E-Op and the semantics of equality we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{x} \Downarrow \mathtt{v}$. Then by E-Var also $\mathtt{E}'(\mathtt{x}) = \mathtt{v}$.
  - Case `L` has the form `L'.l`. Then by Definition 1 also $\mathtt{L}_0$ has the form $\mathtt{L}_0'$`.l` and $\mathtt{E};\mathtt{H} \vdash \mathtt{L}' \Downarrow \mathtt{r}$ and $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0' \Downarrow \mathtt{r}$. We are given stayPrefix(`E`, `H`, `L`) = $\mathtt{C}_L$. Then since solve(`E`, `H`, $\mathtt{C}_L \wedge \mathtt{C}_I \wedge$ `L=v` $\wedge$ `I`, `weak`) = $\mathtt{E}';\mathtt{H}'$ we must have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_L \Downarrow \mathtt{true}$. Similarly we are given stayPrefix(`E`, `H`, `I`) = $\mathtt{C}_I$ so by StayPrefixIdent we have also stayPrefix(`E`, `H`, $\mathtt{L}_0$) = $\mathtt{C}_{L_0}$ where $\mathtt{C}_{L_0}$ is a conjunct within $\mathtt{C}_I$. Then since solve(`E`, `H`, $\mathtt{C}_L \wedge \mathtt{C}_I \wedge$ `L=v` $\wedge$ `I`, `weak`) = $\mathtt{E}';\mathtt{H}'$ we must have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L_0} \Downarrow \mathtt{true}$. So by Lemma 13 and Lemma 12 we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}' \Downarrow \mathtt{r}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0' \Downarrow \mathtt{r}$. We also know $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L=v} \Downarrow \mathtt{true}$, so by E-Op and the semantics of equality we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L} \Downarrow \mathtt{v}$. Then by E-Field we have $\mathtt{H}'(\mathtt{r})$ is a record whose `l` field has value `v`. Then by E-Field also $\mathtt{E}';\mathtt{H}' \vdash \mathtt{r.l} \Downarrow \mathtt{v}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}$.
- Case $k > 0$. Then by Definition 3 there is some neighbor node of $\mathtt{L}_0$'s node that is only $k-1$ hops away from `L`'s node, which contains an l-value $\mathtt{L}_1'$ such that $\mathtt{L}_0'$ `==` $\mathtt{L}_1'$ or vice versa is in `I`, where $\mathtt{L}_0'$ is an alias of $\mathtt{L}_0$ given `E` and `H`. By induction we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_1' \Downarrow \mathtt{v}$. Then since the identities in `I` are satisfied in $\mathtt{E}'$ and $\mathtt{H}'$, by E-IdentityTrue also $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0' \Downarrow \mathtt{v}$. Case analysis on the structure of $\mathtt{L}_0'$:

– Case $\mathtt{L}_0'$ is a variable $\mathtt{x}$. Then by Definition 1 also $\mathtt{L}_0$ is $\mathtt{x}$ and by E-VAR also $\mathtt{E}'(\mathtt{x}) = \mathtt{v}$.
– Case $\mathtt{L}_0'$ has the form $\mathtt{L}'.\mathtt{l}$. Then by Definition 1 also $\mathtt{L}_0$ has the form $\mathtt{L}_{00}.\mathtt{l}$ and $\mathtt{E};\mathtt{H} \vdash \mathtt{L}' \Downarrow \mathtt{r}$ and $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_{00} \Downarrow \mathtt{r}$. We are given stayPrefix(E, H, I) = $\mathtt{C}_I$ so by STAYPREFIXIDENT we have also stayPrefix(E, H, $\mathtt{L}_0'$) = $\mathtt{C}_{L_0'}$ where $\mathtt{C}_{L_0'}$ is a conjunct within $\mathtt{C}_I$. Then since solve(E, H, $\mathtt{C}_L \wedge \mathtt{C}_I \wedge \mathtt{L{=}v} \wedge \mathtt{I}$, weak) = $\mathtt{E}';\mathtt{H}'$ we must have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L_0'} \Downarrow \mathtt{true}$. By the same argument we must also have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L_0} \Downarrow \mathtt{true}$ where stayPrefix(E, H, $\mathtt{L}_0$) = $\mathtt{C}_{L_0}$. So by Lemma 13 and Lemma 12 we have $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}' \Downarrow \mathtt{r}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_{00} \Downarrow \mathtt{r}$. Then since $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0' \Downarrow \mathtt{v}$ by E-FIELD we have $\mathtt{H}'(\mathtt{r})$ is a record whose $\mathtt{l}$ field has value $\mathtt{v}$. Then by E-FIELD also $\mathtt{E}';\mathtt{H}' \vdash \mathtt{r}.\mathtt{l} \Downarrow \mathtt{v}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}$.

□

Lemma 10 *If* stayPrefix(E, H, L) = $\mathtt{C}_L$ *and* stayPrefix(E, H, I) = $\mathtt{C}_I$ *and* $\mathtt{E}'$ *and* $\mathtt{H}'$ *is the relevant update of* E *and* H *for* I *and* L=v *and* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_L \Downarrow \mathtt{true}$ *and and* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_I \Downarrow \mathtt{true}$ *and* $\mathtt{L}_0$ *is a relevant l-value for* I *and* L *given* E *and* H*, then* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}$.

Proof. Case analysis of the structure of $\mathtt{L}_0$:

- Case $\mathtt{L}_0$ is a variable $\mathtt{x}$: By Definition 5 $\mathtt{E}'(\mathtt{x}) = \mathtt{v}$, so the result follows by E-VAR.
- Case $\mathtt{L}_0$ has the form $\mathtt{L}'.\mathtt{l}$: First we argue that stayPrefix(E, H, $\mathtt{L}_0$) = $\mathtt{C}_{L0}$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L0} \Downarrow \mathtt{true}$. If $\mathtt{L}_0$ is L, then these follow from the assumptions of the lemma. Otherwise, by Definition 4 $\mathtt{L}_0$ is an operand in an identity test in I. Then since stayPrefix(E, H, I) = $\mathtt{C}_I$ by STAYPREFIXIDENT we have stayPrefix(E, H, $\mathtt{L}_0$) = $\mathtt{C}_{L0}$, where $\mathtt{C}_{L0}$ is a conjunct in $\mathtt{C}_I$. Then since $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_I \Downarrow \mathtt{true}$ also $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L0} \Downarrow \mathtt{true}$.

  Then by Lemma 12 there is some $\mathtt{r}'$ such that $\mathtt{E};\mathtt{H} \vdash \mathtt{L}' \Downarrow \mathtt{r}'$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}' \Downarrow \mathtt{r}'$. By Lemma 13 and Definition 5 we have $\mathtt{E};\mathtt{H} \vdash \mathtt{r}'.\mathtt{l} \Downarrow \mathtt{v}$ and the result follows by E-FIELD.

□

Lemma 11 *If* stayPrefix(E, H, L) = $\mathtt{C}_L$ *and* stayPrefix(E, H, I) = $\mathtt{C}_I$ *and* $\mathtt{E}'$ *and* $\mathtt{H}'$ *is the relevant update of* E *and* H *for* I *and* L=v *and* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_L \Downarrow \mathtt{true}$ *and and* $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_I \Downarrow \mathtt{true}$ *and* $\mathtt{L}_0$ *is an operand of an identity test in* I *and* $\mathtt{L}_0$ *is not relevant for* I *and* L *given* E *and* H *and* $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$, $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$.

Proof. Case analysis on the structure of $\mathtt{L}_0$:

- Case $\mathtt{L}_0$ is a variable $\mathtt{x}$. Since $\mathtt{x}$ is not relevant, by Definition 5 the value of $\mathtt{x}$ in $\mathtt{E}'$ is the same as that in Euid. Since $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$, by E-VAR we have $\mathtt{E}(\mathtt{x}) = \mathtt{v}_0$, so also $\mathtt{E}'(\mathtt{x}) = \mathtt{v}_0$ and the result follows by E-VAR.
- Case $\mathtt{L}_0$ has the form $\mathtt{L}'.\mathtt{l}$. Since $\mathtt{L}_0$ is an operand in an identity test in I and stayPrefix(E, H, I) = $\mathtt{C}_I$, by STAYPREFIXIDENT we have stayPrefix(E, H, $\mathtt{L}_0$) = $\mathtt{C}_{L0}$, where $\mathtt{C}_{L0}$ is a conjunct in $\mathtt{C}_I$. Then since $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_I \Downarrow \mathtt{true}$ also $\mathtt{E}';\mathtt{H}' \vdash \mathtt{C}_{L0} \Downarrow \mathtt{true}$.

  Therefore by Lemma 12 there is some $\mathtt{r}'$ such that $\mathtt{E};\mathtt{H} \vdash \mathtt{L}' \Downarrow \mathtt{r}'$ and $\mathtt{E}';\mathtt{H}' \vdash \mathtt{L}' \Downarrow \mathtt{r}'$. Since $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_0 \Downarrow \mathtt{v}_0$ by E-FIELD and Lemma 13 we have that $\mathtt{H}(\mathtt{r}').\mathtt{l}$ is $\mathtt{v}_0$. By Definition 5 $\mathtt{H}'(\mathtt{r}')$ also has a field with label $\mathtt{l}$, so by E-FIELD we are done if that field's value is $\mathtt{v}_0$. Suppose not. Then by Definition 5 there is some relevant l-value $\mathtt{L}_1$ of the form $\mathtt{L}_1'.\mathtt{l}$ such that $\mathtt{E};\mathtt{H} \vdash \mathtt{L}_1' \Downarrow \mathtt{r}'$. But then by Definition 1 we have that $\mathtt{L}_0$ and $\mathtt{L}_1$ are aliases so they correspond to the same node in the induced graph of I and L given E and H by Definition 3. But then since $\mathtt{L}_1$ is relevant, by Definition 4 so is $\mathtt{L}_0$ and we have a contradiction.

□

Lemma 12 *If* stayPrefix(E, H, L.f) = C *and* E′;H′ ⊢ C ⇓ true*, then there is some reference* r *such that* E;H ⊢ L ⇓ r *and* E′;H′ ⊢ L ⇓ r*.*

Proof. By StayPrefixField we have L.f = x.l_1.....l_n and $n > 0$ and E;H ⊢ x ⇓ r and E;H ⊢ x.l_1 ⇓ r_1 ⋯ E;H ⊢ x.l_1.....l_{n-1} ⇓ r_{n-1} and C is x=r ∧ x.l_1=r_1 ∧ ⋯ ∧ x.l_1.....l_{n-1}=r_{n-1}.

- Case $n = 1$. Then L is x and C is x=r. Since E′;H′ ⊢ C ⇓ true, by E-Op and the semantics of equality we have E′;H′ ⊢ x ⇓ v_1 and E′;H′ ⊢ r ⇓ v_2 and $v_1 = v_2$. By E-Ref we have that v_2 = r, so the result follows.
- Case $n > 1$. Then L is x.l_1.....l_{n-1}. Since E′;H′ ⊢ C ⇓ true, by E-Op and the semantics of equality we have E′;H′ ⊢ L ⇓ v_1 and E′;H′ ⊢ r_{n-1} ⇓ v_2 and $v_1 = v_2$. By E-Ref we have that $v_2 = r_{n-1}$, so the result follows.

□

Lemma 13 *(Determinism)*

- *If* E;H ⊢ *e* ⇓ v_1 *and* E;H ⊢ *e* ⇓ v_2 *then* $v_1 = v_2$.
- *If stay(*E, *ρ) =* C_1 *and stay(*E, *ρ) =* C_2 *then* $C_1 = C_2$.
- *If stay(*H, *ρ) =* C_1 *and stay(*H, *ρ) =* C_2 *then* $C_1 = C_2$.
- *If stayPrefix(*E, H, *L) =* C_1 *and stayPrefix(*E, H, *L) =* C_2 *then* $C_1 = C_2$.
- *If stayPrefix(*E, H, *I) =* C_1 *and stayPrefix(*E, H, *I) =* C_2 *then* $C_1 = C_2$.

Proof. Straightforward. □

7 Babelsberg/Objects

In this final language, we add support for mutable objects, classes, methods, messages, and inheritance, along with object-oriented constraint definitions (i.e., constraint definitions that can include method calls). This language thus includes all the essential features of actual object constraint languages such as Babelsberg/R, Babelsberg/JS, and Babelsberg/Squeak. It also re-adds records as values that we omitted for simplicity from Babelsberg/UID. The proofs for UID still apply — records as values, when used as value classes, are merely sugar, as explained in Section 7.3.

We can thus now write constraints on the results of message sends, such as the following constraint on the x coordinate of the center of a rectangle:

```
always r.center().x() = 100;
```

The constraint here is on the result of sending the message `center` to `r`, then sending the `x` message to that, and finally sending the `=` message to the result. Depending on how the rectangle is stored, the rectangle's center may well be computed in the `center` method rather than simply being looked up; and even `x` might be computed rather than being stored, for example 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`.

In the earlier Babelsbergs presented in this memo, in the semantics we assume a solver language that is essentially the same as the program language. Constraints that are sent to the constraint solvers could thus be in the program language. This changes for objects. As before, the solver needs to know about uid-records as well as primitive types and i-records; but it does not know about methods or inheritance. So we add a translation in this semantics to inline method calls before passing constraints to the solver.

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 object-oriented language have corresponding primitive types and operations which we can pass directly to the solver. For example, the classes `Integer`, `Float`, and `Boolean` in the source language might be mapped to the primitive integers, floats, and booleans. The semantics includes an automatic boxing and unboxing between instances of these classes and their primitive equivalents. (This boxing and unboxing is part of the formal semantics, but would not necessarily be done in a practical implementation, where we would want primitive types to be represented efficiently.) The `+` method on `Integer` would expand to use the + operation on a primitive representation of itself, and similarly for the `+` method for `Float`, the `or` method for `Boolean`, and so forth. However, in Babelsberg/Objects, in contrast to prior languages, we may no longer mix strings and floats, for example. Previously these both typed as `PrimitiveType`, but now these are full objects that can have different structure (e.g., a `floatvalue` field to store the primitive representation on a `Float` object versus a `stringvalue` field on a `String`). While for these cases, the structural compatibility checks could be made to work (e.g., by using a `primvalue` field for these classes), in general there will be classes in the object-oriented language that don't map to a primitive type; and even for a class that does have a corresponding primitive type, that class can have additional methods that aren't mapped to primitive operations. For example, a `factorial` method for `Integer` has no primitive equivalent and must be translated for the solver.

As with the previous languages, in the semantics, 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 and then set up a constraint between the resulting value and the left hand side. (As with Babelsberg/UID, and in contrast to the simpler languages, this will be an identity constraint rather than an equality constraint.) Otherwise, it must be a method call, which is evaluated by calling the method normally. In addition to any constraint resulting from an assignment statement, each constraint in the constraint store is added to the set of constraints as well, and the resulting set of constraints 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, our execution implicitly gets stuck.

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, it is an error.[2] Otherwise we evaluate the body of the method in a new environment, and return the result. Arguments are passed by value (with pointer semantics), as in most standard object-oriented languages. For instances of primitive types and value classes, the argument can also be copied (since we can't tell the difference between sharing and copying in this case).

The following example shows that passing a variable of primitive type into a method that modifies its argument does not affect the environment outside the method:

```
def addTo(a)
  a := a + 3;
  return a
end
```

Test 32

```
y := 10;
x := addTo(y)
```

After the method returns, `x` will be 13 and `y` will have stayed at 10.

Methods called in the ordinary way can create new constraints, which then persist after the method returns. There is an example in the next section to illustrate this.

Methods can also be invoked by constraint expressions. There are a number of restrictions on methods to allow them to be used in this way — briefly, they can't have side effects, and if they consist of more than just a single return statement they can only be used in the forward direction in constraints. Creating an instance of an ordinary class is regarded as a side effect, so methods that can be invoked by constraint expressions can only create instances of value classes. See Section 7.6 for details, including a discussion of the rationale for this particular choice of restrictions.

If the method can be used in a constraint but only in the forward direction, then it can still be called in the usual way, including passing any parameters by value with pointer semantics. For example, suppose we have a constraint `always c=x.m(a,b)`, and method `m` has multiple statements so that it can only be used in the forward direction, i.e., we can find a value for `c` given values for `x`, `a`, and `b` but not any other direction. In that case, `m` can be called in the usual way, passing `a` and `b` by value with pointer semantics. If the other constraints are such that the correct solution involves finding a value for `b` given values for `c`, `x`, and `a`, a practical system will halt with an error that the constraints are too hard for it to solve, due to a method called from the constraint that can't be inlined (or in the formal semantics, we get stuck). However, in contrast to all the previous examples of constraints that were too hard, the fact that they are too hard is a limitation of the transformations we use for methods called by constraints, rather than being a limitation of the solver.

On the other hand, if method `m` can be used multi-directionally, so that for example we can find a value for `b` given values for `x`, `a`, and `c`, then in the semantics the method is inlined so that the resulting constraints can be turned over to the solver, and we can potentially solve for any of `x`, `a`, `b`, or `c`. This done by creating an environment for method `m` and inlining the returned expression with respect to that environment. The parameters in that environment are constrained to be identical to `a` and `b` respectively, and `self` is constrained to be identical to `x`.

[2] 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. This lookup process isn't represented in our formal semantics since it is standard and doesn't introduce any issues for our focus on constraints.

7.2 Value Classes

Babelsberg/Objects includes *value classes* as well as ordinary, full-featured, classes. The restrictions on value classes make it easier to use them with constraints. A number of existing languages have proposals or support for forms of value classes, such as Java[3] and Scala[4] respectively. In Babelsberg/Objects, instances of value classes are immutable (that is, we need to provide values for all the fields at object creation time, and after that the value class instance cannot be modified).[5] Finally, object identity is not significant for value class instances — we define the identity test method, but it performs a test for equality rather than identity on its argument. However, value classes are more than simple record declarations, since they support methods and inheritance.

In this section, we will use `Point` and `Rectangle` as our example value classes. To distinguish them from ordinary classes, we will create instances just by supplying the arguments, e.g., `Point(10,20)`, whereas instances of ordinary classes will be created using the `new` message, e.g., `Window.new(...)`.

In a practical implementation of Babelsberg, ideally the host language will itself support value classes, so that we can use them directly. If not, we can use ordinary classes, with appropriate conventions about object creation, modification, and not testing for object identity.

Finally, to foreshadow the formal semantics, for simplicity we omit details about object creation, method lookup, and inheritance (which are completely standard), and represent ordinary instances as uid-records and value class instances as i-records.

7.3 Value Classes as Sugar

Value classes are in fact not necessary for Babelsberg — there is a simple transformation that can be used to remove them. However, they are of practical importance, since using them can make Babelsberg/Objects programs much clearer.

To eliminate a use of a value class in a constraint expression, we can instead create a new (ordinary) instance *outside* of the constraint, and in the constraint replace it by appropriate constraints on the attributes of the (new) instance. Consider again a constraint on the center of a rectangle:

```
always r.center() = Point(10,20)
```

We can rewrite this as:

```
point1 := MutablePoint.new(0,0)
always r.center() = point1 && point1.x=10 && point1.y=20
```

If the new value class instance is created using expressions, we need to add appropriate read-only annotations in the rewritten code. For example,

```
always r.center() = Point(d,d+10)
```

is rewritten as:

```
point1 := MutablePoint.new(0,0)
always r.center() = point1 && point1.x=d? && point1.y=d?+10
```

The read-only annotations are necessary since we don't want to satisfy the constraint by changing `d`.

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

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

[5] For use with e.g. distributed systems applications, we would also want to restrict instances of value classes to only hold references to primitive types or to other value class, but this restriction isn't needed for our purposes here.

A remaining issue is that `r.center()` returns a new computed point, which we disallow if points are not instances of a value class, as creating new objects on the heap is considered a side-effect. We can handle this in the following way: when, during our inlining, a constructor is encountered, the object is allocated in a special part of the heap that cannot be reached by user code (using pointer magic or VM introspection or the like). The constructor of the object is not inlined; instead, each argument expression is required to be equal to a field on the new object. This implies limitations on how these objects are constructed, in that their constructors can only assign each argument to a field, in order, and not do any computation. Given these limitations, we will have created a fresh and hidden reference with required equalities in its parts, which makes it effectively immutable. Constraint inlining then proceeds with this object.

Note that the identity of the new (ordinary) instance used in this rewriting cannot escape from the constraint, since ordinary constraint expressions cannot use identity constraints.

Finally, we might create an instance of a value class outside of a constraint expression. These uses of value classes can be replaced by ordinary classes that have no methods that change the state of an instance after it is created, and that override the default identity test method to test for equality instead. For this transformation for statements that create value class instances outside of constraint expressions, we rely on the fact that value class instances cannot be changed after they are created. To illustrate this, consider an example using value classes, and suppose that in fact value class instances *were* mutable.

```
a := Point(10,20);
b := a;
a.x := 30;
```

Just as in Babelsberg/Records, at the end of this program `a=Point(30,20)` but `b=Point(10,20)`. However, if we do the same thing with mutable objects:

```
a := MutablePoint.new(10,20);
b := a;
a.x := 30;
```

then both `a` and `b` have `x=30` (since they are identical).

7.4 Examples

Here is a constraint on the center of a rectangle, where the center is a computed value rather than being stored as an instance variable. (Both `Rectangle` and `Point` are value classes in this example.)

```
r := Rectangle(Point(2,2), Point(10,10));
always r.center() = Point(10,20)
```

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

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

Note that since `Point` is a value class, we can make an instance of it in the constraint expression itself ("`Point(10,20)`"), and also in the `center` method, since we are doing point addition and division in that method. (If it had been an ordinary class, creating such an instance would have been a side effect, disallowing this constraint.)

The constraint is evaluated in the following way. The code for Rectangle's `center` method is found, and inlined to construct an equality constraint between the result of evaluating

`(self.upper_left + self.lower_right) / 2` and a new point with `x=10`, `y=20`. Constructing the center point from the first expression, as well as looking up and inlining the `=` method on it, explodes into a network of simpler constraints, all required, that can then be handed to the solver in one conjunction. Note how the local names are translated into a global solver environment by appending a digit to every duplicate name.

$$
\begin{array}{ll}
 & \text{// constraint for the receiver of the center method} \\
\text{required} & self1 = \{upper\_left : \{x : 2,\ y : 2\},\ lower\_right : \{x : 10,\ y : 10\}\} \\
 & \text{// the point addition called from the center method} \\
\text{required} & self2 = self1.upper\_left \\
\text{required} & arg1 = self1.lower\_right \\
 & \text{// the point constructor called from the point addition method} \\
\text{required} & x1 = self2.x + arg1.x \\
\text{required} & y1 = self2.y + arg1.y \\
 & \text{// point division by scalar called from center method} \\
\text{required} & self3 = \{x : x1,\ y : y2\} \\
\text{required} & arg2 = 2 \\
 & \text{// point constructor called from point division method} \\
\text{required} & x2 = self3.x \,/\, arg2 \\
\text{required} & y2 = self3.y \,/\, arg2 \\
 & \text{// point constructor for the static point} \\
\text{required} & x3 = 10 \\
\text{required} & y3 = 20 \\
 & \text{// and for the point equality} \\
\text{required} & self4 = \{x : x2,\ y : y2\} \\
\text{required} & arg3 = \{x : x3,\ y : y3\} \\
\text{required} & self4.x = arg3.x \wedge self4.y = arg3.y
\end{array}
$$

As mentioned, for each variable name in a scope a unique global variable name is created. The solver works only on the global names. During evaluation, each local name maps to a global name, which in turn maps to a value. Entering the center method creates a local environment in which `self` is bound to the rectangle by value. A global alias `self1` is created and constrained to refer to that same value. Similarly, the global alias `self2` is a point, namely the upper left corner of the mutable rectangle. This alias is then used when we explode the `+` message to that point. `self3` is also a point, namely the new point that is returned by the point addition method and now used as receiver of the scalar division method on points. For methods that take arguments, global names for the argument names are created and constrained similarly. For example, `arg1` is the global name in this exploded constraint for the argument to the `+` method for `Point`. Thus, the solver never has to deal with different scopes or name clashes.

Now consider a similar example but using mutable rectangles and points on the heap, using the translation when using their constructors in constraints as defined in Section 7.3.

$$\begin{array}{lr}
 & \text{// stays for the heap} \\
\text{required} & \mathtt{H}(r) = \{upper_left : p_{ul},\ lower_right : p_{lr}\} \\
\text{required} & \mathtt{H}(r_{ul}) = \{x : x1,\ y : y1\} \\
\text{required} & \mathtt{H}(r_{lr}) = \{x : x2,\ y : y2\} \\
\text{required} & p_{ul} = r_{ul} \\
\text{required} & p_{lr} = r_{lr} \\
\text{required} & x1 = 2 \\
\text{required} & y1 = 2 \\
\text{required} & x2 = 10 \\
\text{required} & y2 = 10 \\
 & \text{// the center method} \\
\text{required} & self1 = r \\
 & \text{// the point addition called from the center method} \\
\text{required} & self2 = \mathtt{H}(self1).upper_left \\
\text{required} & arg1 = \mathtt{H}(self1).lower_right \\
 & \text{// the point constructor called from the point addition method} \\
\text{required} & \mathtt{H}(r_c) = \{x : x3,\ y : y3\} \\
\text{required} & self3 = r_c \\
\text{required} & x3 = \mathtt{H}(self2).x\ +\ \mathtt{H}(arg1).x \\
\text{required} & y3 = \mathtt{H}(self2).y\ +\ \mathtt{H}(arg1).y \\
 & \text{// point division by scalar called from center method} \\
\text{required} & self4 = self3 \\
\text{required} & arg2 = 2 \\
 & \text{// point constructor called from point division method} \\
\text{required} & \mathtt{H}(r_d) = \{x : x4,\ y : y4\} \\
\text{required} & self5 = r_d \\
\text{required} & x4 = \mathtt{H}(self4).x\ /\ arg2 \\
\text{required} & y4 = \mathtt{H}(self4).y\ /\ arg2 \\
 & \text{// point constructor for the static point} \\
\text{required} & \mathtt{H}(r_s) = \{x : x5,\ y : y5\} \\
\text{required} & self6 = r_s \\
\text{required} & x5 = 10 \\
\text{required} & y5 = 20 \\
 & \text{// and for the point equality} \\
\text{required} & self7 = self5 \\
\text{required} & arg3 = self6 \\
\text{required} & \mathtt{H}(self7).x = \mathtt{H}(arg3).x \wedge \mathtt{H}(self7).y = \mathtt{H}(arg3).y
\end{array}$$

Before we inline the methods, we create appropriate stays on the heap. Entering the center method creates a local environment in which `self` is bound to `r`, which is a reference to the mutable rectangle on the heap. We create aliases for the receivers and arguments as before, but now treat constructors specially, in that we create a new reference on the heap, bind it to a new `self` alias, and use the arguments to the constructor as constraints on the parts of the object, in effect turning the assignment of these arguments to the parts of the newly created object into required equality constraints. This means we execute constructors in a sort of "mixed mode," in which heap objects are created as needed and their arguments are turned into equality constraints on their fields. This implies limitations on the constructors, namely that they be just methods binding arguments to the fields of the newly created object, without any control structures.

This example also illustrates why we need to solve for object identity and type before values — if `r` had been an instance of a class with a different implementation of the `center` method, it would have been exploded in a different way.

Now consider an example with mutable rectangles (instances of `MutableRectangle`) but points again as instances of a value class. If we assign to the rectangle's `upper_left`, the solver will update `lower_right` to keep the center at `Point(10,20)`:

Test 33
```
r := new MutableRectangle(Point(2,2), Point(10,10));
always r.center() = Point(10,20);
r.upper_left := Point(100,2);
```

We can also accomplish the same thing using a `once` constraint:

Test 34
```
r := new MutableRectangle(Point(2,2), Point(10,10));
always r.center() = Point(10,20);
once r.upper_left.x = 100
```

Finally, this test fails because the `always` and `once` constraints are incompatible:

Test 35
```
r := new MutableRectangle(Point(2,2), Point(10,10));
always r.center() = Point(10,20);
once r.center().x = 100
```

Here is another example of using a method in a constraint. Suppose we add a `double` method to `Float`, and then use it in a constraint:

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

Test 36
```
x := 0;
y := 0;
always y=x.double();
y := 20;
```

After the program runs, `y` will be 20 and `x` will be 10.

For this program, the constraint `y=x.double()` explodes into the following network:

required $self1 = x$
required $y = 2 * self1$

This conjunction of constraints works equally well for determining `x` or determining `y`, even though `double` is actually a message to `x`.

As noted above, methods called in the ordinary way can create new constraints, which then persist after the method returns. Here is a simple example. Suppose we have a `BankAccount` class that includes a `balance` method. We can then define the following method:

```
def require_min_balance(acct,min)
  always acct.balance >= min?
end;
```

Now suppose we call `require_min_balance(a,10)` on some account `a`. (This is just calling the method from an ordinary statement, either at the top level or some other method, *not* from a constraint expression.) Thereafter the balance in the account `a` must be at least 10 (a persistent constraint). We use a `?` annotation on the variable `min` to make it read-only with respect to this constraint — otherwise the solver would be free to change either the account balance or the local variable `min`. Section 3 includes a discussion of read-only

annotations; as noted earlier, we do not include rules for read-only annotations in the semantics, but they would be straightforward to add.

Note the consequences of our calling convention for such methods. Consider:

Test 37

```
a := BankAccount.new;
m := 10;
require_min_balance(a,m);
m := 100
```

Even though we reassigned `m`, the minimum balance for the account stays at 10, since we passed `m` (a primitive type) by value. Of course, we are passing the account `a` by value as well:

```
a := BankAccount.new;
require_min_balance(a,10);
a := BankAccount.new;
```

After the second assignment, `a` is bound to a new account — but the minimum balance constraint is on the previous instance of `BankAccount` (which will presumably be garbage collected since there aren't any references to it).

If we do want a constraint that continues to be enforced even if the account or the minimum is rebound, we can instead write a method that is a minimum balance test and use it in a constraint:

```
def has_min_balance(acct,min)
  return acct.balance >= min;
end
```

Test 38

```
a := BankAccount.new;
m := 10;
always has_min_balance(a,m?);
m := 100;
```

Programmers will need to be aware of the different semantics for these two cases, and select the appropriate variant when it makes a difference. Note the placement of the read-only annotation on the variable `m` in the `always` constraint — it would make no sense to place it on `min` in the `has_min_balance` method, since that is just a method, not itself a constraint.

Implementing such methods requires that the semantics (and implementations) keep the local variables in such a method invocation as long as the constraints it creates are active. Here is a somewhat artificial variant of the original `require_min_balance` method to illustrate this:

```
def require_min_balance_with_locals(acct,min)
  b := acct.balance;
  always b = acct.balance;
  always b >= min?;
end;
```

We could also add a minimum balance constraint in the initialization method for `BankAccount`. This version requires that the initial balance be provided, but has a default of 0 for the minimum balance in case it isn't provided.

```
def init(initial_balance,min=0)
  self.balance := initial_balance;
  always self.balance >= min?;
end;
```

7.5 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 compatibility 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.

7.6 Additional Restrictions on Constraint Expressions

Methods can of course have side effects, but such methods can't be used in the expressions that define constraints. (This is one of the restrictions noted in Section 4.1 on requirements for constraint expressions.) This subsection describes some consequences of this restriction when we have objects, methods, and object identity. There are also some additional restrictions on methods called from constraints that are used in other than the forward direction, discussed below.

Regarding side effects in methods, consider the `pop` method for a class `Stack`, which has a side effect as well as returning the value popped from the stack. This program fragment is OK:

```
x := stack.pop;
```

This works, because the RHS of the assignment is evaluated first, and then we set up a `once` constraint using the value popped from the stack. (Thus the RHS of an assignment can have side effects, in contrast to `always` or `once` constraints.)

In contrast, these program fragments are not OK:

```
once x == stack.pop;
```

```
always x == stack.pop;
```

It's easy to see why we don't want to allow this: we should be able to evaluate the constraint expression as a test, and if it evaluates to true the constraint is satisfied. This would just not work in these stack examples.

As noted previously, creating an instance of an ordinary class is regarded as a side effect, so methods that can be invoked by constraint expressions can only create instances of value classes and not ordinary classes.[6]

Creating a constraint is a kind of side effect, at least potentially, so another consequence of the prohibition on side effects in constraint expressions is that methods called from a constraint expression cannot themselves create other constraints, or call further methods that do so. The following program is not allowed, and illustrates side effects resulting from adding constraints in methods called from other constraints.

```
def test(i)
  always medium i=5;
  return i+1;
end
```

Test 39

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

[6] As also noted previously (Footnote 1), in a practical implementation the programmer might be able to make cautious use of benign side effects in a constraint expression. Such benign side effects might include creating instances of ordinary classes, for example, constructing a temporary instance of an ordinary class that is garbage collected before it is visible outside the constraint. In the formal semantics, however, we simply disallow side effects in constraint expressions.

Suppose we are using a least-squares solver that supports soft constraints. Every time we evaluate the expression `y=test(x)`, we add another medium constraint that `x=5`. Since we are finding a least squares solution, this nudges `x` more toward 5 each time, diminishing the influence of the `x=10` constraint.[7]

Here are two other issues with constraints that call methods that add constraints, for the record in case we do want to try to allow this in some restricted set of circumstances:

First, we would need to solve their constraints eagerly rather than just accumulating the constraints and solving all at once. Here's an example that demonstrates this.

```
def test(x)
  y := 11;
  always y=x;
  if y>10 then return x else return 11;
end;
a := 5;
always b=test(a);
```

The expected outcome is that this method always returns a value larger than `10`. If we don't solve eagerly, however, `y` is 11 and `x` is `5` when we evaluate the `if` statement, so when used in a constraint, this method would in this case return `5`.

Another reason why nested constraints are disallowed is because the method could add soft constraints, and itself be invoked by a soft constraint — for example, suppose that in the above program we instead had `always weak b=test(a)`. We would need an algebra for combining priorities, for example, take the minimum of the priorities.

Methods called by constraints can assign to variables, but users should be careful with assignments that would be visible outside the method. A valid use-case for global assignment is caching, but it would produce unpredictable results if, for example, a global counter were incremented and returned on each call.

If there are assignments, the method can only be used in the forward direction in constraints that call it (i.e., to compute the result given the inputs). The only way that statements in a method other than the final return aren't dead code is if they have side effects, so for practical purposes we can restate this restriction as follows: for a method to be used in other than just the forward direction in constraints that call it, the method must consist of a single `return` statement.

It would be possible to relax this restriction somewhat, so that other kinds of methods could be used in reverse in constraints, but the resulting restrictions would be harder to express, and would have an *ad hoc* feel, likely making it harder for the programmer to keep track of them. So in the design presented here we use a simple, easily understood restriction that covers an important set of practical cases. See Appendix A.7 for a description of a backward compatibility mode for methods that accommodates some methods with multiple statements, which is currently implemented in Babelsberg/Ruby.

Here are some examples.

Consider an iterative `sum` method for the class `Array`. This works as expected for normal imperative code.

```
def sum()
  ans := 0;
  i := 0;
  while i<self.length
```

[7] This issue only arises with soft constraints and a global comparator. With constraints that are required, re-adding a required constraint is idempotent. Or if we use a local comparator, re-adding the soft constraint will also not change the resulting solution. But in any case, adding a constraint is conceptually different from testing whether it is satisfied.

```
    ans := ans + self[i];
    i := i+1;
  end;
  return ans;
end
```

We can also use it in the forward direction in a constraint:

```
a := Array.new(2);
a[0] := 10;
a[1] := 20;
s := 30;
always s = a.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, the method doesn't work backwards — for example, we can't constrain the sum of the array and expect the system to update one or more elements to satisfy the constraint. So the constraint in the last line below will be too hard for the system to solve:

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

We *can* provide something that works both forward and backward by writing an ordinary method that sets up the appropriate network of addition constraints. For this to work correctly, this method should take an argument that holds the sum of the array and that is mutable. We will use an instance of a class `Cell` with a `value` method that can be used to read or constrain the cell's value. The following `sum_equal` method sets up an appropriate network of constraints.

```
def sum_equal(array,sum)
  helper(array,0,sum);
end;

def helper(array,start,sum)
  if start>=array.length() then always sum.value()=0;
    else
       partial := Cell.new(0);  /* create a local variable */
       helper(array,start+1,partial);
       always sum.value()=array[start]+partial.value();
  end if;
end;
```

Now our example works:

```
a := Array.new(2);
a[0] := 10;
a[1] := 20;
sum_equal(a,Cell.new(50));
```

If we subsequently set `a[1]` to 5, `a[0]` will become 45 to keep the constraints satisfied.

Note that in contrast to the earlier version, the constraints are not automatically re-applied if `a` is reassigned;

the programmer needs to call `sum_equal` again.

Needing to call `sum_equal` explicitly if `a` is reassigned, as well as needing to use instances of `Cell` to hold the sum, is slightly annoying. We can address both of these issues by making `sum` be an instance variable of arrays, and setting up the necessary network of constraints in the array's `initialize` method. (We're still using `Cell` in the helper method, but it's not visible outside that method.)

```
def initialize()
  c := Cell.new(0);
  always c.value()=self.sum();
  helper(self,0,c);
end;
```

As an aside, we expect that we will discover additional useful design patterns (like the pattern for constructing multi-way constraints on collections) as we make further use of the Babelsberg implementations.

It is tempting to write `sum` as a method that returns the sum and that still can be used multi-directionally, perhaps:

```
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;
```

Unfortunately, this doesn't meet the restriction that methods called from constraints consist only of a single return statement. Worse, if we tried inlining it, the method would expand infinitely. So, at least for now, `sum_equal` seems like it is as good as we can do.

A related example is a method for `Array` that tests whether the argument is one of the elements of that array.

```
def contains(x)
  i:=0;
  while i<self.length && self[i]!=x
    i:=i+1;
  end;
  return i<self.length;
end;
```

Since this includes more than one statement, it can only be used in a constraint in the forward direction:

```
a := Array.new(10);
a.fill(100);
c := false;
always c = a.contains(5);
a[3] := 5;
```

The analogous method to `sum_equal`, say `must_contain`, unfortunately would not be very useful: since we don't have Prolog-style backtracking, it would probably just be satisfied in the reverse direction by setting

the first element of the array.

7.7 Identity Constraint Examples

The following example illustrates how assignment for objects with object identity is handled. This uses an example class `MutablePoint`, which is an ordinary class whose instances can be changed and that have individual identities, in contrast to the value class `Point` used previously.

Test 40

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

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:

Test 41

```
p := MutablePoint.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`. The final constraint on `q.x` is unsatisfiable. 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 getting stuck due to an unsatisfiable required constraint.

While cloning a mutable 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;
```

However, this variant, in which we make the constraints on the window's widths strong rather than required, is OK. (We don't create a new window to satisfy the strong constraints on the widths.)

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

As a point of comparison, this restriction on not creating new objects isn't relevant for Babelsberg/Records or when we use instances of value classes in Babelsberg/Objects. Here is one of the above examples using value classes:

```
p := Point(10,20);
```

```
q := p;
always p.x=5;
always q.x=10;
```

This is fine, and at the end of the program, `p` and `q` will refer to two different value class instances.

There can be explicit identity constraints, as in Babelsberg/UID. Again as in Babelsberg/UID, identity constraints must be satisfied when they are first created. For example:

Test 42

```
x:= Window.new(....);
y := x;
always y==x;
x := Circle.new(....);
```

Here `x` and `y` are identical at the time the identity constraint `always y==x` is created. And given this identity constraint, after the last assignment both `x` and `y` refer to the newly created circle. This example also illustrates how type changes must originate with an assignment statement, but can ripple further via explicit identity constraints.

Identity constraint can also be created in methods, but that only makes sense if we pass real objects (with pointer semantics):

```
def make_identical(a,b)
  always a == b
end
```

Test 43

```
x:= Window.new(....);
y := x;
make_identical(x,y);
x := Circle.new(....)
```

To illustrate some effects of passing arguments to methods in constraints, here are a few more examples.

```
def make_equal_to_5(x)
  always x = 5
end
```

Test 44

```
a := 0;
make_equal_to_5(a)
```

Here a is passed by-value, so the constraint is not visible outside the method and has no effect on `a`.

```
def make_a_equal_b_plus_3(a,b)
  always a = b + 3
end
```

Test 45

```
x := 0;
y := 0;
make_a_equal_b_plus_3(x,y);
x := 10;
y := 10
```

Again, due to the by-value semantics, this has no effect on `x` and `y`.

Suppose we have a method that asserts a constraint on an object:

```
def pt_x_equals_5(pt)
  always pt.x = 5
end
```

If we use a value object that lives on the stack, the constraint has no effect on the passed point, and `q` stays at $(0, 0)$:

Test 46
```
q := Point(0,0);
pt_x_equals_5(q)
```

However, if we pass a mutable point that lives on the heap, we pass it by-reference, and the constraint does affect `q`:

Test 47
```
q := MutablePoint.new(0,0);
pt_x_equals_5(q)
```

After this code runs, `q.x` is `5`.

7.8 Formalism

7.8.1 Syntax

The syntax is augmented to support method invocation as expressions. We introduce syntax for the method body. Additionally, we allow creating new objects as expressions now. We also have a form of immutable records and consider them as "value objects," which can respond to methods. Finally, we introduce the `nil` value. The entire syntax is given below.

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

We assume that the set of variable names ranged over by metavariable `x` includes the name `self`.

7.8.2 Semantics

Since we have methods with local scopes now, we introduce a global $\mathbb{E}$ that maps global variable names to values. The rules for execution still work on a local environment, but a local environment is only a mapping from local names to globally unique names now. Method lookup creates new local environments,

and constraints are translated to translate local variable names to global variable names for the solver. The solver still returns a global environment, and does not know about the local environments and their mapping into the global environment. In addition, the constraint stores `I` and `C` are replaced with $\mathbb{I}$ and $\mathbb{C}$, respectively. These now store constraints in pairs with the local environment they were created in. For readability, we now write $\mathbb{H}$ instead of `H`, so all the global stores are written in the same font.

$$\boxed{\mathit{lookup}(\mathtt{v},\mathtt{l}) = (\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{b})}$$

"Lookup of method `l` in the object or value `v` returns the formal parameter names $\mathtt{x}_1$ through $\mathtt{x}_n$ and the method body `b`"

This judgment is opaque: our semantics does not depend on how method lookup is performed.

$$\boxed{\mathit{enter}(\mathbb{E},\mathtt{E},\mathbb{H},\mathbb{C},\ \mathbb{I},\mathtt{v},\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{e}_1 \cdots \mathtt{e}_n) = (\mathbb{E}',\mathtt{E}_m,\mathbb{H}',\mathbb{C}',\mathbb{I}')}$$

"Invoking a method on `v` with argument names $\mathtt{x}_1$ through $\mathtt{x}_n$ and arguments $\mathtt{e}_1$ through $\mathtt{e}_n$ constructs the method scope $\mathtt{E}_m$ and may update the heap and constraint stores."

This is a helper judgment that simplifies the definition of evaluation for method calls (shown below).

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\texttt{|}\mathbb{H}_1\texttt{|}\mathbb{C}_1\texttt{|}\mathbb{I}_1\texttt{|}\mathtt{v}_1\texttt{>} \\ \cdots \\ \texttt{<}\mathbb{E}_{n-1}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}_{n-1}\texttt{|}\mathbb{C}_{n-1}\texttt{|}\mathbb{I}_{n-1}\texttt{|}\mathtt{e}_n\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|}\mathtt{v}_n\texttt{>} \\ \texttt{<}\mathbb{E}_n\texttt{|}\mathtt{E}_m\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|self := v>} \longrightarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathtt{E}_0\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{>} \\ \texttt{<}\mathbb{E}_0\texttt{|}\mathtt{E}_0\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|}\mathtt{x}_1\texttt{ := }\mathtt{v}_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}_{n+1}\texttt{|}\mathtt{E}_1\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{>} \\ \cdots \\ \texttt{<}\mathbb{E}_{2n-1}\texttt{|}\mathtt{E}_{n-1}\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|}\mathtt{x}_n\texttt{ := }\mathtt{v}_n\texttt{>} \longrightarrow \texttt{<}\mathbb{E}_{2n}\texttt{|}\mathtt{E}_n\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{>}\end{array}}{\mathit{enter}(\mathbb{E},\mathtt{E},\mathbb{H},\mathbb{C},\ \mathbb{I},\mathtt{v},\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{e}_1 \cdots \mathtt{e}_n) = (\mathbb{E}_{2n},\mathtt{E}_n,\mathbb{H}_n,\mathbb{C}_n,\mathbb{I}_n)} \quad (\textsc{Enter})$$

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

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

$$\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{c}\texttt{>} \Downarrow \texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{c}\texttt{>} \quad (\text{E-}\textsc{Const})$$

$$\frac{\mathtt{E}(\mathtt{x}) = \mathtt{x}_g \qquad \mathbb{E}(\mathtt{x}_g) = \mathtt{v}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{x}\texttt{>} \Downarrow \texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{v}\texttt{>}} \quad (\text{E-}\textsc{Var})$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{r}\texttt{>} \qquad \mathbb{H}'(\mathtt{r}) = \{\mathtt{l}_1\texttt{:}\mathtt{v}_1\texttt{,}\ldots\texttt{,}\mathtt{l}_n\texttt{:}\mathtt{v}_n\} \qquad 1 \le i \le n}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e.l}_i\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{v}_i\texttt{>}} \quad (\text{E-}\textsc{Field})$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\{\mathtt{l}_1\texttt{:}\mathtt{v}_1\texttt{,}\ldots\texttt{,}\mathtt{l}_n\texttt{:}\mathtt{v}_n\}\texttt{>} \qquad 1 \le i \le n}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e.l}_i\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{v}_i\texttt{>}} \quad (\text{E-}\textsc{ValueField})$$

$$\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{r}\texttt{>} \Downarrow \texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{r}\texttt{>} \quad (\text{E-}\textsc{Ref})$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|v}_1\texttt{>} \quad \texttt{<}\mathbb{E}_0\texttt{|E|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|e}_2\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v}_2\texttt{>} \quad \texttt{v}_1 \ [\![\oplus]\!] \ \texttt{v}_2 = \texttt{v}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1 \ \oplus \ \texttt{e}_2\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v>}} \quad \text{(E-Op)}$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|v>} \quad \texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|e}_2\texttt{>} \Downarrow \texttt{<E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v>}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1 \ \texttt{==} \ \texttt{e}_2\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|true>}} \quad \text{(E-IdentityTrue)}$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|v}_1\texttt{>} \quad \texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|e}_2\texttt{>} \Downarrow \texttt{<E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v}_2\texttt{>} \quad \texttt{v}_1 \neq \texttt{v}_2}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1 \ \texttt{==} \ \texttt{e}_2\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|false>}} \quad \text{(E-IdentityFalse)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|v>} \\ \textit{lookup}(\texttt{v},\texttt{l}) = (\texttt{x}_1 \cdots \texttt{x}_n,\texttt{s; return e}) \\ \textit{enter}(\mathbb{E}_0,\texttt{E},\mathbb{H}_0,\mathbb{C}_0,\ \mathbb{I}_0,\texttt{v},\texttt{x}_1 \cdots \texttt{x}_n,\texttt{e}_1 \cdots \texttt{e}_n) = (\mathbb{E}_1,\texttt{E}_m,\mathbb{H}_1,\mathbb{C}_1,\mathbb{I}_1) \\ \texttt{<}\mathbb{E}_1\texttt{|E}_m\texttt{|}\mathbb{H}_1\texttt{|}\mathbb{C}_1\texttt{|}\mathbb{I}_1\texttt{|s>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{>} \\ \texttt{<}\mathbb{E}'\texttt{|E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{|v}_r\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e.l(e}_1\texttt{,...,e}_n\texttt{)>} \Downarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{|v}_r\texttt{>}} \quad \text{(E-Call)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}_0\texttt{|}\mathbb{C}_0\texttt{|}\mathbb{I}_0\texttt{|v>} \\ \textit{lookup}(\texttt{v},\texttt{l}) = (\texttt{x}_1 \cdots \texttt{x}_n,\texttt{return e}) \\ \textit{enter}(\mathbb{E}_0,\texttt{E},\mathbb{H}_0,\mathbb{C}_0,\ \mathbb{I}_0,\texttt{v},\texttt{x}_1 \cdots \texttt{x}_n,\texttt{e}_1 \cdots \texttt{e}_n) = (\mathbb{E}_1,\texttt{E}_m,\mathbb{H}_1,\mathbb{C}_1,\mathbb{I}_1) \\ \texttt{<}\mathbb{E}_1\texttt{|E}_m\texttt{|}\mathbb{H}_1\texttt{|}\mathbb{C}_1\texttt{|}\mathbb{I}_1\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v}_r\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e.l(e}_1\texttt{,...,e}_n\texttt{)>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|v}_r\texttt{>}} \quad \text{(E-CallSimple)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\texttt{|}\mathbb{H}_1\texttt{|}\mathbb{C}_1\texttt{|}\mathbb{I}_1\texttt{|v}_1\texttt{>} \\ \cdots \\ \texttt{<}\mathbb{E}_{n-1}\texttt{|E|}\mathbb{H}_{n-1}\texttt{|}\mathbb{C}_{n-1}\texttt{|}\mathbb{I}_{n-1}\texttt{|e}_n\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|v}_n\texttt{>} \\ \mathbb{H}_n(\texttt{r})\uparrow \qquad \mathbb{H}' = (\mathbb{H}_n \bigcup \{(\texttt{r},\ \{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\})\})\end{array}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|new } \{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|r>}} \quad \text{(E-New)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\texttt{|}\mathbb{H}_1\texttt{|}\mathbb{C}_1\texttt{|}\mathbb{I}_1\texttt{|v}_1\texttt{>} \\ \cdots \\ \texttt{<}\mathbb{E}_{n-1}\texttt{|E|}\mathbb{H}_{n-1}\texttt{|}\mathbb{C}_{n-1}\texttt{|}\mathbb{I}_{n-1}\texttt{|e}_n\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|v}_n\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}_n\texttt{|}\mathbb{C}_n\texttt{|}\mathbb{I}_n\texttt{|}\{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}\texttt{>}} \quad \text{(E-Value)}$$

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

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

We typecheck with respect to the global environments $\mathbb{E}$. Method calls do not typecheck: even though we allow method calls in expressions and thus in constraints syntactically, our rules for creating constraints given below inline method invocations. Identity constraints also do not typecheck: they are solved separately and should not appear in ordinary constraints (see the rules for statement evaluation below). Finally, new (non-value) object construction does not typecheck, since constraints must be side-effect-free.

Types distinguish between primitive values and objects, and for objects the type keeps track of their fields:

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

$$\mathbb{E};\mathbb{H} \vdash \mathtt{c} : \mathtt{PrimitiveType} \tag{T-Const}$$

$$\frac{\mathbb{H}(\mathtt{r})=\mathtt{o} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{o} : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{r} : \mathtt{T}} \tag{T-Ref}$$

$$\frac{\mathbb{E}(\mathtt{x}) = \mathtt{v} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{v} : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{x} : \mathtt{T}} \tag{T-Var}$$

We now transform all local variables to their global names using the inlining rules given below before passing them to the solver. Thus, only global variable names type.

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e} : \{\mathtt{l}_1{:}\mathtt{T}_1, \ldots, \mathtt{l}_n{:}\mathtt{T}_n\} \qquad 1 \leq i \leq n}{\mathbb{E};\mathbb{H} \vdash \mathtt{e}.\mathtt{l}_i : \mathtt{T}_i} \tag{T-Field}$$

We add a typing rule for dereferences, although these defererences will only appear in expressions that have been translated by our inlining judgment. This ensures that the inlined expressions still typecheck, and simplifies the rules for constraint solving.

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e} : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{H(e)} : \mathtt{T}} \tag{T-Deref}$$

Since we now re-added records (as instances of value classes), and we want to be able to perform operations (at least equality) on them, we just require the operands of an operation to be of the same type, not necessarily a primitive type.

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e}_1 : \mathtt{T} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{e}_2 : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{e}_1 \ \oplus \ \mathtt{e}_2 : \mathtt{PrimitiveType}} \tag{T-Op}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e}_1 : \mathtt{T}_1 \cdots \mathbb{E};\mathbb{H} \vdash \mathtt{e}_n : \mathtt{T}_n}{\mathbb{E};\mathbb{H} \vdash \{\mathtt{l}_1{:}\mathtt{e}_1, \ldots, \mathtt{l}_n{:}\mathtt{e}_n\} : \{\mathtt{l}_1{:}\mathtt{T}_1, \ldots, \mathtt{l}_n{:}\mathtt{T}_n\}} \tag{T-ValueObject}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e} : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \rho \ \mathtt{e}} \tag{T-Priority}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{C}_1 : \mathtt{T} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{C}_2 : \mathtt{T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{C}_1 \ \wedge \ \mathtt{C}_2} \tag{T-Conjunction}$$

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

As before we assume that the solver natively supports records and uninterpreted functions (which we use to represent the heap). We do not assume, however, that the solver understands methods, which can now be part of constraint expressions. This requires us to essentially inline methods before passing constraints to the solver.

$\boxed{\text{stay}(\texttt{x=v},\ \rho) = \texttt{C}}$

$\boxed{\text{stay}(\texttt{r=o},\ \rho) = \texttt{C}}$

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

$\boxed{\text{stay}(\mathbb{H},\ \rho) = \texttt{C}}$

The rules for creating stay constraints are simply a combination of the UID rules plus the rule for stay constraints on value classes from Babelsberg/Records, adapted to this formalisms environment and heap.

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

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

$$\frac{\texttt{x}_1 \text{ fresh} \cdots \texttt{x}_n \text{ fresh} \qquad \text{stay}(\texttt{x}_1\texttt{=v}_1,\ \rho) = \texttt{C}_1 \cdots \text{stay}(\texttt{x}_n\texttt{=v}_n,\ \rho) = \texttt{C}_n}{\text{stay}(\texttt{x = \{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\texttt{\}},\ \rho) = (\rho\ \texttt{x = \{l}_1\texttt{:x}_1\texttt{,...,l}_n\texttt{:x}_n\texttt{\})}\ \wedge\ \texttt{C}_1\ \wedge \cdots \wedge\ \texttt{C}_n} \qquad \text{(StayRecord)}$$

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

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

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

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

"Inlining expression `e` from the local environment `E` turns into expression $\texttt{e}'$. To connect variables across method calls, the constraint expression $\texttt{e}_C$ is returned."

We use an inlining judgment to translate expressions into a representation suitable for the solver. This combines the previous stayPrefix(`E`, `H`, `L`) = `C` and $\texttt{E;H} \vdash \texttt{C} \rightsquigarrow \texttt{C}'$ judgments from Babelsberg/UID. In particular, we translate local variables into their names in the global environment and provide a semantics for method calls inside constraints. Arguments to method calls are constrained to be equal to the expression that generated them. Inlining does not allow updates to the heap, so no new heap is returned. We do allow assignments to locals in inlined methods, however, so the global environment can change as a result of inlining.

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

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

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

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

The I-Field rule expresses the property of the `E;H` $\vdash$ `C` $\rightsquigarrow$ `C'` judgment from Babelsberg/UID that each expression of the form `e.l`, where `e` evaluates to a heap reference `r`, is translated into `H(e').l` (recursively translating `e` into `e'` through further inlining).

$$\frac{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{>} \quad \texttt{<}\mathbb{E}'\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\{\texttt{l}_1\texttt{:v}_1,\ldots,\texttt{l}_n\texttt{:v}_n\}\texttt{>}}{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{.l}\texttt{>}} \quad \text{(I-ValueField)}$$

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

$$\frac{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_1\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_{C_a},\texttt{e}_a\texttt{>} \qquad \texttt{<}\mathbb{E}',\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_{C_b},\texttt{e}_b\texttt{>}}{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_1\ \oplus\ \texttt{e}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_{C_a}\wedge\texttt{e}_{C_b},\texttt{e}_a\ \oplus\ \texttt{e}_b\texttt{>}} \quad \text{(I-Op)}$$

$$\frac{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_1\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_{C_a},\texttt{e}_a\texttt{>} \qquad \texttt{<}\mathbb{E}',\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_{C_b},\texttt{e}_b\texttt{>}}{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}_1\ \texttt{==}\ \texttt{e}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_{C_a}\wedge\texttt{e}_{C_b},\texttt{e}_a\ \texttt{==}\ \texttt{e}_b\texttt{>}} \quad \text{(I-Identity)}$$

The I-Identity rule includes the StayPrefixIdent rule from Babelsberg/UID, but now works on arbitrary expressions. Through I-Field and I-ValueField, all but the last part are forced to stay as they are. Note: In contrast to the previous `E;H` $\vdash$ `C` $\rightsquigarrow$ `C'` judgment, we no longer translate `==` to `=`, but instead assume that the solver treats it the same. This ensures that inlined identity constraints still do not type (in S-Once), even if both operands have the same type.

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v>} \\ \texttt{<}\mathbb{E}_0\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v}_1\texttt{>} \\ \cdots \\ \texttt{<}\mathbb{E}_{n-1}\texttt{|E|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e}_n\texttt{>} \Downarrow \texttt{<}\mathbb{E}_n\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v}_n\texttt{>} \\ \texttt{e}_C = (\texttt{e=v}\ \wedge\ \texttt{e}_1\texttt{=v}_1\ \wedge\ \cdots\ \wedge\ \texttt{e}_n\texttt{=v}_n) \\ \mathit{lookup}(\texttt{v},\texttt{l}) = (\texttt{x}_1\ \cdots\ \texttt{x}_n\texttt{,s; return e}) \\ \mathit{enter}(\mathbb{E}_n\texttt{,E},\mathbb{H}_n\texttt{,}\mathbb{C}_n,\ \mathbb{I}_n\texttt{,v,x}_1\ \cdots\ \texttt{x}_n\texttt{,e}_1\ \cdots\ \texttt{e}_n) = (\mathbb{E}'\texttt{,E}_m\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}) \\ \texttt{<}\mathbb{E}'\texttt{|E}_m\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|s>} \longrightarrow \texttt{<}\mathbb{E}''\texttt{|E'|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{>} \\ \texttt{<}\mathbb{E}''\texttt{|E'|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'''\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v}_r\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\texttt{E},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l(e}_1\texttt{,}\ldots\texttt{,e}_n\texttt{)>} \rightsquigarrow \texttt{<}\mathbb{E}'''\texttt{,e}_C\texttt{,v}_r\texttt{>}} \quad \text{(I-Call)}$$

Methods that have any statements at all can only be used in a one-way manner. This is ensured by evaluating the return expression and using only the value in the constraint. Because we are retranslating all constraints on each semantic step, this return value will get updated when its dependencies change, it just won't work in the other direction.

When inlining a method with more than one statement, the statements are simply executed. In particular, this means that we eagerly choose which branch of if-statements to inline and eagerly unroll loops. Further, the I-Call rule above and the I-MultiWayCall rule below ensure that methods being used in constraints have no side effects. This is accomplished by requiring the initial heap to remain unchanged.

Similarly, methods used in constraints cannot declare nested constraints; this is accomplished by requiring the initial sets of ordinary and identity constraints to remain unchanged.

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

For methods that only return an expression, we inline the expression and pass it to the solver. Note that we execute the argument expressions and receiver for their value (potentially executing through other methods), and also inline them, potentially executing the same methods twice (once through I-Call and once through E-Call.) Although not ideal in terms of providing the cleanest possible semantics, in practical terms this should not be a problem, because we prohibit side-effects in these methods.

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

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

"Re-inlining the constraint store $\mathbb{C}$ returns a constraint C"

"Re-inlining the constraint store $\mathbb{I}$ returns a constraint C"

We use the below judgments to re-translate all constraints in the store using our inlining rules before solving.

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

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

$$\frac{\mathbb{C}_0 = \mathbb{C} \setminus \{(\mathtt{E}, \rho\, \mathtt{e})\} \qquad \texttt{<}\mathbb{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{I}\texttt{,}\mathbb{C}_0\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_0\texttt{,}\mathtt{C}_0\texttt{>} \quad \texttt{<}\mathbb{E}_0\texttt{,E,}\mathbb{H}\texttt{,}\mathbb{C}_0\texttt{,}\mathbb{I}\texttt{,e>} \rightsquigarrow \texttt{<}\mathbb{E}'\texttt{,}\mathtt{e}_{C_e}\texttt{,e'>}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{I}\texttt{,}\mathbb{C}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'\texttt{,}\mathtt{C}_0 \ \wedge\ \rho\ \texttt{(e'} \ \wedge\ \mathtt{e}_{C_e}\texttt{)>}} \text{ (I-ReinlineC)}$$

$$\frac{\mathbb{I}_0 = \mathbb{I} \setminus \{(\mathtt{E}, \texttt{required e})\} \qquad \texttt{<}\mathbb{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}_0\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_0\texttt{,}\mathtt{C}_0\texttt{>} \quad \texttt{<}\mathbb{E}_0\texttt{,E,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}_0\texttt{,e>} \rightsquigarrow \texttt{<}\mathbb{E}'\texttt{,}\mathtt{e}_{C_e}\texttt{,e'>}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'\texttt{,}\mathtt{C}_0 \ \wedge\ \texttt{required (e'} \ \wedge\ \mathtt{e}_{C_e}\texttt{)>}} \text{ (I-ReinlineI)}$$

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

This is a helper judgment for use in evaluating assignment statements and identity constraints. These statements are the only ones that can cause the types of variables to change. It combines both phases of solving into one rule. These steps cannot occur separately, and the typechecking that in Babelsberg/UID was done in between phases now needs to use a possibly modified global environment, so we have combined both phases into this one judgment for simplicity. Otherwise, the steps and phases are exactly equivalent: in

the first phase, we propagate the new equality constraint through all existing identity constraints in order to update other variables and fields as needed. In the second phase we solve all of the non-identity constraints, plus any constraints C created by inlining identity constraints, and we typecheck under the new environment before solving the value constraints.

$$\frac{\begin{array}{c}\text{stay}(\mathbb{E}, \texttt{weak}) = \mathtt{C}_{E_s} \qquad \text{stay}(\mathbb{H}, \texttt{weak}) = \mathtt{C}_{H_s} \qquad \texttt{<}\mathbb{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_i\texttt{,}\mathtt{C}_i\texttt{>} \\ \mathbb{E}';\mathbb{H}' \models (\mathtt{C}_i \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_0 \wedge \mathtt{C}_{H_s} \wedge \mathtt{e}_1\texttt{=}\mathtt{e}_2) \\ \text{stay}(\mathbb{E}', \texttt{required}) = \mathtt{C}_{E'_s} \qquad \text{stay}(\mathbb{H}', \texttt{required}) = \mathtt{C}_{H'_s} \qquad \texttt{<}\mathbb{E}'\texttt{,}\mathbb{H}'\texttt{,}\mathbb{I}\texttt{,}\mathbb{C}\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_c\texttt{,}\mathtt{C}\texttt{>} \\ \mathbb{E}_c;\mathbb{H}' \vdash \mathtt{C} \qquad \mathbb{E}'';\mathbb{H}'' \models (\mathtt{C} \wedge \mathtt{C}_0 \wedge \mathtt{C}_{E'_s} \wedge \mathtt{C}_{H'_s} \wedge \mathtt{e}_1\texttt{=}\mathtt{e}_2)\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}_1 \texttt{ == } \mathtt{e}_2\texttt{|}\mathtt{C}_0\texttt{>} \Longrightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}''\texttt{>}} \quad \text{(TwoPhaseUpdate)}$$

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

The stepping rules are refactored to work with the local environments and the new constraint and identity-constraint stores.

$$\frac{\begin{array}{c}\mathtt{E}(\mathtt{x})\uparrow \qquad \mathbb{E}(\mathtt{x}_g)\uparrow \\ \texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{v}\texttt{>} \\ \mathtt{E}' = \mathtt{E} \bigcup \{(\mathtt{x}, \mathtt{x}_g)\} \qquad \mathbb{E}' = \mathbb{E} \bigcup \{(\mathtt{x}_g, \mathtt{v})\}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{x} \texttt{ := } \mathtt{e}\texttt{>} \longrightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathtt{E}'\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{>}} \quad \text{(S-AsgnNewLocal)}$$

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

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{v}\texttt{>} \\ \texttt{<}\mathbb{E}'\texttt{,}\mathtt{E}\texttt{,}\mathbb{H}'\texttt{,}\mathbb{C}'\texttt{,}\mathbb{I}'\texttt{,}\mathtt{e}_l\texttt{.l>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,}\mathtt{e}_c\texttt{,}\mathtt{e}'\texttt{>} \\ \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{e}'\texttt{==}\mathtt{v}\texttt{|}\mathtt{e}_c\texttt{>} \Longrightarrow \texttt{<}\mathbb{E}'''\texttt{|}\mathbb{H}''\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}_l\texttt{.l} \texttt{ := } \mathtt{e}\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'''\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{>}} \quad \text{(S-AsgnLValue)}$$

Note that the above rule can only be used in constraint-construction mode if $\mathbb{H}$=$\mathbb{H}'$. This effectively restricts assignments in constraints to work on values. Note also, that assignments in constraints are just executed, but do not set up a required equality constraint between the LHS and RHS. (The current Babelsberg/Ruby includes a "backwards compatibility mode" which sets up required equality constraints when assignment occurs, but it's not clear whether we should retain support for that. See Appendix A.7.)

We also do not allow the use of the following rules for `once` and `always` in constraint-construction mode, because the inlining rules disallow updating the constraint store and heap. As discussed previously (Section 7.6), in a practical implementation we might want to support benign side effects in methods that are invoked by constraint expressions, including methods that themselves create new constraints; but this is not modeled by this semantics.

$$\frac{\begin{array}{cc}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}_0\texttt{>} \Downarrow \texttt{<}\mathbb{E}_0\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{v}\texttt{>} & \texttt{<}\mathbb{E}_0\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}_1\texttt{>} \Downarrow \texttt{<}\mathbb{E}_1\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{v}\texttt{>} \\ \texttt{<}\mathbb{E}_1\texttt{,}\mathtt{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,}\mathtt{e}_0\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_2\texttt{,}\mathtt{e}_{C_0}\texttt{,}\mathtt{e}'_0\texttt{>} & \texttt{<}\mathbb{E}_2\texttt{,}\mathtt{E}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,}\mathtt{e}_1\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}_3\texttt{,}\mathtt{e}_{C_1}\texttt{,}\mathtt{e}'_1\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\texttt{once } \mathtt{e}_0 \texttt{ == } \mathtt{e}_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}_3\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{>}} \quad \text{(S-OnceIdentity)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\texttt{once } \mathtt{e}_0 \texttt{ == } \mathtt{e}_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{>} \\ \mathbb{I}' = \mathbb{I} \bigcup \{(\mathtt{E}, \mathtt{e}_0 \texttt{ = } \mathtt{e}_1)\}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\texttt{always } \mathtt{e}_0 \texttt{ == } \mathtt{e}_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}'\texttt{>}} \quad \text{(S-AlwaysIdentity)}$$

$$\frac{\begin{array}{c}\mathtt{C}_0 = \rho\ \mathtt{e} \qquad \mathtt{<\mathbb{E},E,\mathbb{H},\mathbb{C},\mathbb{I},e>} \rightsquigarrow \mathtt{<\mathbb{E}',e_{C_e},e'>} \qquad \mathtt{C}_0' = \rho\ (\mathtt{e}' \wedge \mathtt{e}_{C_e}) \\ \mathbb{E}';\mathbb{H} \vdash \mathtt{C}_0' \qquad \mathrm{stay}(\mathbb{E}', \mathtt{required}) = \mathtt{C}_{E_s} \qquad \mathrm{stay}(\mathbb{H}, \mathtt{required}) = \mathtt{C}_{H_s} \\ \mathtt{<\mathbb{E}',\mathbb{H},\mathbb{I},\mathbb{C}>} \rightsquigarrow \mathtt{<\mathbb{E}'',C>} \qquad \mathbb{E}''';\mathbb{H}' \models (\mathtt{C} \wedge \mathtt{C}_{E_s} \wedge \mathtt{C}_{H_s} \wedge \mathtt{C}_0')\end{array}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|once\ C_0>} \longrightarrow \mathtt{<\mathbb{E}'''|E|\mathbb{H}'|\mathbb{C}|\mathbb{I}>}} \quad \text{(S-Once)}$$

$$\frac{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|once\ C_0>} \longrightarrow \mathtt{<\mathbb{E}'|E|\mathbb{H}'|\mathbb{C}|\mathbb{I}>} \quad \mathbb{C}' = \mathbb{C} \bigcup \{(\mathtt{E}, \mathtt{C}_0)\}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|always\ C_0>} \longrightarrow \mathtt{<\mathbb{E}'|E|\mathbb{H}'|\mathbb{C}'|\mathbb{I}>}} \quad \text{(S-Always)}$$

$$\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|skip>} \longrightarrow \mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}>} \quad \text{(S-Skip)}$$

$$\frac{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|s_1>} \longrightarrow \mathtt{<\mathbb{E}'|E'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'>} \quad \mathtt{<\mathbb{E}'|E'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|s_2>} \longrightarrow \mathtt{<\mathbb{E}''|E''|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|s_1;s_2>} \longrightarrow \mathtt{<\mathbb{E}''|E''|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}} \quad \text{(S-Seq)}$$

$$\frac{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|e>} \Downarrow \mathtt{<\mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|true>} \quad \mathtt{<\mathbb{E}'|E|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|s_1>} \longrightarrow \mathtt{<\mathbb{E}''|E'|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|if\ e\ then\ s_1\ else\ s_2>} \longrightarrow \mathtt{<\mathbb{E}''|E'|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}} \quad \text{(S-IfThen)}$$

$$\frac{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|e>} \Downarrow \mathtt{<\mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|false>} \quad \mathtt{<\mathbb{E}'|E|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|s_2>} \longrightarrow \mathtt{<\mathbb{E}''|E'|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|if\ e\ then\ s_1\ else\ s_2>} \longrightarrow \mathtt{<\mathbb{E}''|E'|\mathbb{H}''|\mathbb{C}''|\mathbb{I}''>}} \quad \text{(S-IfElse)}$$

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

$$\frac{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|e>} \Downarrow \mathtt{<\mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|false>}}{\mathtt{<\mathbb{E}|E|\mathbb{H}|\mathbb{C}|\mathbb{I}|while\ e\ do\ s>} \longrightarrow \mathtt{<\mathbb{E}'|E|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'>}} \quad \text{(S-WhileSkip)}$$

References

[1] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[2] 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.

[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] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In *Proceedings of the 1994 European Conference on Object-Oriented Programming*, pages 260–279, July 1994.

[13] 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.

A Appendix

Sections A.1 – A.3 of this appendix discuss some issues that are important in practical implementations of Babelsberg but that are mostly orthogonal to the task at hand of providing a formal semantics for Babelsberg. The remaining sections include descriptions of some design alternatives that were considered and later dropped, but that seem promising enough that we might come back to them some day.

A.1 Warnings and Debugging

We don't deal with warnings and debugging in the formal semantics for Babelsberg. In a production implementation, however, including such support would be key.

First, all of the places where the formal semantics gets stuck to model an error state should have explicit, understandable error messages in a real implementation. A few additional issues:

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. We may want to issue a warning in this case in a practical implementation. (This would be a simple case to check for.)

As described previously, in the interests of predictability, the system will not create a new (standard) object to satisfy constraints. There should be an appropriate error message in such a case if the unsatisfied constraints are required, and a warning if they are soft.

A.2 Read-Only Annotations

First, here is some discussion of how read-only annotations work. Intuitively, when choosing a best solution to a set of constraints, 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 [13]. A *one-way constraint* can be represented by annotating all but one of the constrained variables as read-only.

Here's a simple example of the effect of a read-only annotation on a variable. First consider these constraints:

required	$x = y$
strong	$x = 3$
weak	$y = 4$

This has the solution $x \mapsto 3, y \mapsto 3$, since the strong constraint trumps the weak one.

But suppose we add a read-only annotation:

required	$x = y$?
strong	$x = 3$
weak	$y = 4$

Because y is read-only in the $x = y?$ constraint, the solver can't use that constraint in determining the value of y, and so the solution this time is $x \mapsto 4, y \mapsto 4$.

Further, if we make the constraint $x = 3$ be required:

required	$x = y?$
required	$x = 3$
weak	$y = 4$

the constraints now have no solution: on the one hand, any solution must satisfy the required $x = 3$ constraint, but on the other, we still can't use $x = y?$ in determining a value for y. (This is a so-called "blocked" set of constraints [3].)

The constraint semantics presented in [3] and as discussed so far accounts for read-only annotations on variables, but not expressions. In practical languages, programs are often more concise if we also permit read-only annotations on expressions. To accommodate these, the implementations do a simple rewrite of a read-only expression $e?$ by introducing a fresh variable t, replacing $e?$ in the original constraint with $t?$, and adding a new required constraint $t = e$. For example, given these constraints:

strong	$x = (y + z + 5)?$
medium	$x = 3$
weak	$y = 0$
weak	$z = 0$

we would rewrite the first constraint using a new variable t:

strong	$x = t?$
required	$t = y + z + 5$
medium	$x = 3$
weak	$y = 0$
weak	$z = 0$

The solution is $x \mapsto 5, y \mapsto 0, z \mapsto 0$, as desired.

This extension is particularly useful for expressions involving method calls in Babelsberg/Objects, e.g., an expression like `rect.center()?`, but works for the simpler Babelsbergs as well.

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, if necessary 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 useful to include a solver for geometric constraints, either in 2 or 3 dimensions.

Some extensions that already exist in the practical languages are a solver (and constructs in the solver language) for finite domain constraints, as in the clpfd library for SWI Prolog. Other additional solvers are available for strings and local propagation constraints, as solved for example by DeltaBlue. Supporting the latter required 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, when processing a constraint 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. This works even if the programmer does not specify a propagation function, because DeltaBlue has a default propagation method for equality (and some other relations, such as scaling.)

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 The Perturbation Model vs. the Refinement Model

A key issue in integrating constraints with imperative programming is how to represent variables that change over time. Earlier versions of Kaleidoscope [7, 8, 9] used a *refinement model*, in which ordinary variables were represented as a stream of "pellucid variables," each holding a value at a different time. This was in turn adapted from Lucid [1]. Later versions of Kaleidoscope [12] used a *perturbation model*, in which variables were represented conventionally. Assignment perturbed the value of a variable, and then the constraints took over and adjusted the values of other variables so that they were re-satisfied if necessary. The formal semantics described here uses a variation of the perturbation model, which we believe is cleaner. (In particular, rather than just perturbing the value of a variable by changing it, we model assignment as a `once` constraint between the variable and the value of the expression on the right hand side of an assignment statement.)

The refinement model does provide additional capabilities — for example syntax for referring to both the current and previous values of a variable — but at a cost, both conceptually for the programmer and also for the language implementor. At least for the common cases, we believe the refinement model and our current model provide the same answers. Consider again the example in Section 4 of unsatisfiable constraints arising from the following program:

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

This behavior we modeled with the rules for Babelsberg/PrimitiveTypes is the same as that of the refinement model, in which the program would be equivalent to these constraints:

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

The read-only annotations in the refinement model serve the same role as do the evaluation rules in our current model. In the example, in our current semantics we model the final assignment `x:=x+15` by first evaluating `x+15` in the old environment, and then adding a required `once` constraint that `x` be equal to that value. In the refinement model, we model the final assignment as $x_2 = x_1? + 15$, where the read-only annotation on x_1 accomplishes the same thing, by preventing the solver from changing x_1 to satisfy this constraint even though the other constraint that gives x_1 the value of 5 is only weak. Similarly, stay constraints in the current semantics are modeled as weak constraints equating the variable with its current value; in the refinment model these are weak constraints relating the variables representing the current and previous versions, with the previous version annotated as read-only.

A.5 Issues with Using Once Constraints in the Semantics for Assignments

In all of the Babelsbergs, in the semantics we model an assignment by evaluating the right-hand-side, creating a new variable if needed for the left-hand-side, setting it to the new value, and then solving the set of constraints that include a `once` constraint that the left-hand-side equal that value. This lets assignment interact correctly with other constraints in the constraint store, if any. For example, as a consequence of this semantics, the following program will get stuck in the formal semantics, or raise an exception in a practical implementation:

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

However, there are some subtle issues in connection with this representation.

The analogous program still gets stuck, as one would want, in Babelsberg/Records or when using value classes, since records and instances of value classes are immutable.

```
x := {a:0};
always x.a=10;
x := {a:5};
```

But once we get to Babelsberg/UID and Babelsberg/Objects with ordinary objects, the situation changes. Consider the analogous program in Babelsberg/UID:

```
x := new {a:0};
always x.a=10;
x := new {a:5};
```

Here, the right-hand-side of the last assignment is evaluated, the reference to the new record is assigned to `x`, and then the system immediately changes its `a` field to 10 — it is only the reference that is immutable, not the record on the heap. This is arguably weird, and so one of the design alternatives we considered was to make the right hand side be read-only (recursively). However, this might be computationally expensive, and would also cause difficulties when creating circular structures (see below). So in the current design, we just evaluate the right-hand-side to a value, but don't do anything further about making it recursively read-only.

A.5.1 Circular Structures

There is an issue regarding creating circular structures if we convert assignment statements to `once` constraints, with the value on the right hand side recursively annotated as read only. (The same issue arose in an earlier version of the semantics in which we simply turned an assignment into a `once` constraint with the right hand side annotated as read only, instead of first evaluating the right hand side.) 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 would be a bit strange. But this solution doesn't require any changes to the formal semantics, and addresses all the previous issues.

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`.

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.)

A.6 Alternatives to Structural Compatibility Checks

Our model includes structural compatibility checks on constraints, which prevent the solver from potentially generating new kinds of records and other odd behaviors. But since these ideas seem to keep resurfacing, we outline some of our rejected alternatives in this subsection, in case they end up being useful later.

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

If the solver will need to compare two records with different fields, 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. (Such a constraint might be unsatisfied either because the values in one or more corresponding fields weren't equal, or were of different types, or because the corresponding field didn'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}` or `{x:5}={x:"squid"}`? We need to determine an error for additional or missing fields, and for comparing different types and this error needs to be such that it can be meaningfully compared with the error for values of fields that hold numeric types. 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 without filtering out weird cases with structural compatibility checks is that it may result in less predictability for programmers. Consider:

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

If this isn't rejected by a structural compatibility check, there there are a number of plausible values that the solver could find for `p` and `q` — namely `{x:0}`, `{y:1}`, and `{x:0,y:1}` — and unclear which is better. The non-determinism affects the structure of the result rather than just its value, and would prevent achieving Goal 2 in our list in Section 3.2.

Yet another complication is that we would want a minimality condition on records. Suppose we let an `always` constraint create a new record if need be:

```
always 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 untamed by structural compatibility checks, led us to reject the alternative outlined in this subsection.

For Babelsberg/Objects, the analog of this behavior would be to allow the constraint solver generate new classes on the fly. Particularly since there would be multiple possible classes, as before, this seems overly complex and unpredictable.

A.7 Backwards Compatibility Mode for Methods

Methods called by constraints must be free of side effects. (In a practical language, benign side effects might be allowed, but we don't model this in the formal semantics.) This seems like a reasonable restriction that we shouldn't attempt to change, although we probably ought to make precise what constitutes a benign side effect.

In addition, if a method consists of more than just a single return statement, it can only be used in the forward direction in constraints. This provides a straightforward way for both the programmer and the implementor to understand how this should be handled: if the method is used in the forward direction only, it can be called in the ordinary way; if it is potentially used multi-directionally, it is exploded.

If we are trying to add constraints to an existing language with a substantial class library, this restriction does imply that many methods, even though they are side-effect-free, can only be used in the forward direction. The Babelsberg/Ruby implementation provides a "backwards compatibility mode" that relaxes the restriction in an attempt to make more of these existing methods be usable multi-directionally. (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 mode is incompatible with the new semantics for Babelsberg described in this memo. We describe it here for completeness; updating it to be compatible with the new semantics (or simply removing it) is left for future work.

The Babelsberg/Ruby implementation uses two interpreter modes: standard imperative execution mode and constraint construction mode. When methods are evaluated in constraint construction mode, assignment statements are converted to two-way equality constraints. However, we've now decided that methods called from a constraint expression cannot themselves create other constraints, or call further methods that do so (Section 7.6), which rules out this transformation in its current form.

For the record, though, if we relaxed this restriction, we might want to support this conversion. It still doesn't seem entirely clean, so we should probably still 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, can be 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;
```

Note though that the semantics of conditional statements are unchanged — we still eagerly evaluate the test and select the appropriate branch (with no backtracking to the other branch).

If we also had a way to explode methods on demand to support recursion, we might also be able to automatically convert methods with loops to recursions that could then be used multi-directionally. 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. (Again, we still eagerly evaluate the test in the conditional, i.e., we evaluate the original `while` statement a definite number of times — this method won't let us try different array lengths to satisfy the constraints.)

As we've been noting, these transformations won't work in all cases. As an extreme example, if we have an `encrypt` method that takes a plaintext message `m` and a public key `k`, we are unlikey to be able to run the method backwards to find the original message given the cyphertext. Coming up with a good set of transformations, a specification of when they will work, and whether the result covers a useful set of cases, is an open problem. Certainly they work for methods consisting just of a sequence of assignment statements and a final return statement, but this seems like not that useful a class of methods to run backwards.

For the record, here are some more straightforward examples of when the tranformation doesn't work.

Consider an absolute value method for integers:

```
def abs()
  if self>=0 then return self else return 0-self;
end;
```

Suppose we use it in this program:

```
x := 5;
y := 5;
always y=x.abs();
always x<0;
```

This works up until the final statement. However, when we try to satisfy `x<0`, because of the semantics of the `if` statement (which eagerly evaluates the test, with no backtracking), we incorrectly conclude the constraints are unsatisfiable.

Instead, the `abs` method as written should only be used in the forwards direction; if we want one that also works backwards, we should write it as a disjunction that can be turned over to the solver (or add support for Prolog-style backtracking — see the next section).

Another case that doesn't work arises when the method has a conditional, and the two branches return different types. Even if these are both value classes, this still doesn't explode correctly using the current rules. For example, suppose that in addition to standard points, we have a class `ZeroPoint` representing a point with its x and y both 0 (which then avoids storing those fields). Also suppose that both kinds of points implement an `optimize` method that returns a new object, which is the point represented in the optimal way. For ordinary points, the method is:

```
def optimize()
  if self.x=0 && self.y==0 then return ZeroPoint() else return self;
end;
```

For `ZeroPoint`, `optimize` just returns `self`.

Now consider this code that uses the `optimize` method:

```
p := Point(5,5);
q := Point(5,5);
always q = p.optimize();
```

If we implement this by exploding the `optimize` method, the existing transformations don't work since `q` might be either a `Point` or a `ZeroPoint`. We might be able to get it to work in this case, but the transformations become more complex and probably more fragile, and it's not clear that the complexity is worth the price.