Felgentreff,
Babelsberg

Object-Constraint Programming
Ein Untertitel ist Optional

The Long Title of the Work to Be Done
This is Optionally Optional

von

Tim Felgentreff

Dissertation
zur Erlangung des akademischen Grades des

Doktor der Naturwissenschaften
(Doctor rerum naturalium)

vorgelegt
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam.

Betreuer

Prof. Dr. Robert Hirschfeld

Fachgebiet Software-Architekturen
Hasso-Plattner-Institut
Universität Potsdam

20. Mai 2015

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertation selbst angefertigt und nur die im Literaturverzeichnis aufgeführten Quellen und Hilfsmittel verwendet habe.

Diese Dissertation oder Teile davon wurden nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht.

Ich versichere weiterhin, dass ich diese Arbeit oder eine andere Abhandlung nicht bei einer anderen Fakultät oder einer anderen Universität eingereicht habe

Potsdam, den 20. Mai 2015

Tim Felgentreff

Abstract

The english abstract.

Zusammenfassung

Die Zusammenfassung auf deutsch.

Contents

List of Figures

List of Tables

List of Listings

Part I.

Solving Constraints on Object Behavior

1. Introduction

Constraints occur in a variety of application domains relevant to computer science. Constraints are declarative: they specify a desired relation that should hold, but not how to achieve this. Constraint programming is the paradigm that allows developers to express constraints directly and have a constraint solver take care of how to reach and then maintain the desired state. Constraint programming thus provides flexibility in the solving approach, and avoids scattered code to ensure that constraints are observed throughout the execution of the program.

Domains where constraints are relevant include graphics, physical simulations, planning, or interactive puzzles. Some examples for these are: that two graphical objects keep a minimum distance when moved or change their appearance when the mouse moves over them; that a piston stays at a constant distance to a crank or that a resistor obey Ohm's law; that a video streaming service adjust the quality of the stream to play smoothly based on the load on the network; or that a game of Sudoku may only be solved according to the rules. In these domains, it is often easier to state *what* the rules are than *how* to achieve them. In the example of a Sudoku, constraints allow to fully define the possible inputs simply by adding constraints on the cells that their input be between one and nine, and that each number must appear exactly once in each row, column, and sub-block.

Declarative constraints offer advantages over an imperative, object-oriented programming style. The solving strategy may be exchanged without having to adapt the constraint declarations. Some problems can be naturally expressed using constraints, and the code is thus a close to the problem domain. Finally, constraint solvers have a clean semantics when it comes to combining multiple and possibly competing constraints. Due to these advantages, many constraint-programming systems have been proposed over the years, notable systems include Sketchpad [**sutherland-ifips-1963**] in 1963, ThingLab [**borning-toplas-1981**] in 1981, and —>

There have been numerous attempts to provide these advantages as part of general purpose object-oriented programming languages. These attempts usually chose one of two main alternatives for incorporating constraints into object-oriented programs: The most common way is as a library, which has the advantage that no changes are required to the underlying programming language and that the ways in which the solving process can affect the imperative runtime are trivially known. However, this kind of integration leaves it up to the programmer to translate the relations between objects into constraints for the solver and back at appropriate times in the program. The alternative technique is to integrate constraints directly into the underlying programming language, and provide either an extended syntax or a Domain-specific Language (DSL) for specifying them. In

cite

this case, there is the cost of having to support a new or extended programming language.

fix this argument, its stupid, we have the same problem

Despite these advantages, we see multiple reasons why constraints have failed to become a commonly used tool in the utility belt of object-oriented programming languages. Even where constraints may be applied directly, purely imperative programming is still the norm. We believe this is in part due to performance considerations, the lack of integration with existing, imperative code, as well as inexperience of the programmers with the features and limits of constraint solvers.

First, although very efficient solving strategies such as incremental solving or edit constraints exist, these often have to be applied explicitly or performance will suffer. In contrast, modern imperative language runtimes apply a number of optimizations automatically — either at compile-time or using Just-in-Time (JIT) compilation techniques — so that even unoptimized code provides good performance. A goal of this work is thus to use applicable techniques to improve the performance of the solving process in general, and make solver specific features to improve performance, such as incremental resolving, more accessible.

cite

Second, previous integration attempts of constraints into object-oriented languages did not unify the constructs for encapsulation and abstraction from both paradigms, making it hard to re-use code written in one style within code written in the other. Prior work on constraint DSLs or fully integrated languages requires programmers to essentially learn two languages. Abstraction mechanisms in the object-oriented language — i.e., methods and classes — are not available in the constraint-oriented code, which uses its own abstractions. When crossing the border from imperative to constraint code, constraints can break encapsulation to access fields of objects directly, and thus limit the re-usability of constraint code across objects with similar interfaces, but different internal structure. Another goal of this work is thus to unify the abstraction mechanisms in a way that respects encapsulation and makes code re-use easier in semantically clean way.

Third, while it may be easy to formulate a problem as a constraint, it is sometimes very hard to formulate it in a way that a solver can efficiently find a solution. In general, it is much easier to state constraints than to solve them — some problems are extremely difficult, and others are undecidable. For example, while it is easy to ask that $a^n + b^n = c^n$ for three integers, a solver can only conclude that this is unsatisfiable . Constraint satisfaction is an NP-hard Boolean Satisfiability Problem (SAT), but if we restrict the classes of constraints to a useful subset in a particular domain, quite efficient solvers are available. Many practical solvers impose such restrictions on the kinds of constraints they can solve, for example, by supporting only floats and not integers or requiring numeric equalities and inequalities to be linear. For example, the Cassowary solver [**badros-tochi-2001**] can efficiently solve multi-way linear equations on floats using the simplex method; Z3 [**de2008z3**] is an satisfiability modulo theories (SMT) can numerically solve for reals, integers, booleans, as well as record data types; Kodkod [**torlak2007kodkod**], Z3, and Ilog [**puget1994c++**, **ilogmanual**] can enumerate solutions; and DeltaBlue [**freeman-benson-phoenix-1989**] can solve multi-way

cite

constraints using local propagation. An even more restricted approach is to only consider one-way constraints that can compute an output value given new inputs, but not vice versa (e.g., [**hudson-uist-1996**, **myers-ieee-computer-1990**]). A third goal of this work is thus to allow multiple solvers to cooperate to find a solution to a given problem, thus making efficient constraints solving applicable to a wider range of practical applications.

soft constraints?

HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH

We observe a shift in how new programming languages are positioned in the last decades of computer science. While the major paradigms, such as OO, FP, or declarative stay the same, more recently, compund paradigms were proposed, such as CIP CLP, that attempt to combine the benefits of different paradigms. The reason is obvious: there is not one paradigm that works well for all use cases.

Think of natural programming — we have many techniques that work well for specific use-case, like oo for models of the real world, constraints to express problems and invariants, stepping to let the computer to integration for us, pattern matching to let it find a needle in a haystack or decide on the matching reaction to an input,... However, looking at the statistics, one might argue that OO has won, for its benefits in modelling the real world and interactive processes, but with the drawbacks that when another paradigm may be more appropriate, we revert to using it through libraries or as an entirely separate language or DSL. This discourages their use, and as a consequence, workarounds and "best practices" proliferate that attempt to express something imperatively that may be more easily expressed, for example, in constraints. All to avoid adding yet another library or DSL.

We argue that there are two areas where this line of research must move forward. On the one hand, we need new programming systems that are interactive, explorable, and acknowledge that programming is about more than writing and compiling strings into an executable form. The other is, how we can take the advantages and lessons we learned in the past decades and combine them to create not new, but integrated programming languages, where the choice of paradigm is not inherent in the language, but different paradigms are available and integrated in a way so programmers can move between paradigms as they see fit. We are already on the way for both of these paths. This work is a modest step on the latter path.

HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH

1.1. Challenges

The aforementioned goals of this work present a number of challenges.

Constraint Solvers are Non-deterministic "Magic"

note
cite
cite
icte?

A major challenge for constraint systems has been to find functions to choose good solutions. The theoryis that using constraint programming is the holy grail: the user states a problem, and the computer solves it. In practice, humans use a lot more implicit context when searching for solutions to problems. This context is not only implicit, it is in parts not even consciously known. Thus, when the problem is put to the computer, many implicit constraints are omitted, allowing the system to come up with surprising and undesirable solutions.

The fact that constraint solvers are often provided as black boxes without good debugging support exacerbates this problem. While imperative execution encodes the order of operations in its semantics, declarative programming semantics is more or less independent of the actual order of execution. While this has a number of advantages (for example, declarative languages can generally more easily make scale to use multiple threads of execution), if something does not work as intended the gap between actual execution and user code is vast.

cite
cot
cite
cite
cite

Specialized constraint systems are used for type checking and inference, or checking of models of object-oriented systems against the execution of their implementation. These system can infer many interesting properties and notify the programmer when an assumption cannot be proven. However, use cases such as acceptability oriented computingor input fuzzingnot only check, but also attempt to correct execution of programs when assumptions fail.

When a constraint solver is allowed to change the execution state to satisfy a set of constraints, implicit preferences need to be weighed: should the solver be allowed to change the types of variables or even the semantics of the language to satisfy the constraint? Is the identity of an object important, or can the solver substitute another that satisfies the constraints? Which constraints should lead to a run-time error when they are violated, and which constraints can be safely ignored?

Object-oriented Imperative Programming is the Norm

needed

Considering the most popular languages according to , object-orientation has arguably won. At the very least, we can infer from such data that a large number of developers are at least familiar with object-oriented programming style. Furthermore, since a large number of useful code is written in object-oriented languages, any new language has to consider how to interface with existing work. If the conceptual distance to object-like modules and imperative operation is too far, abstractions do not match, and it is hard to interface with existing code, and even harder to understand how systems are working together. This deters from using constraints when it makes sense, because they may make the system harder to understand and maintain.

Many problems indeed are more naturally expressed with a step-wise definition in an imperative style. Constraint languages have no notion of mutable state

or sequences of events, and problems involving user interaction are thus hard to express in a purely constraint-declarative style. However, when a problem naturally lends itself to constraints, using a solver can reduce the size of the user program significantly and make the code more flexible and understandable. Thus, we argue, there is a benefit to be gained from integrating the two paradigms and making both options available to a larger number of programmers.

For many languages, foreign-function interfaces (FFIs) exist to interface with the underlying system and other libraries, and many constraint solving libraries use such a mechanism to offer constraint solving in general purpose programming languages. However, these libraries are often restricted to the lowest common denominator, which are the primitive values for which a simple mapping between the solver and the language exists.

cite, at least Z3py

cite

In this work we aim towards a deeper integration of these two paradigms, to enable constraints in a wide variety of existing OO applications, by re-using the same abstractions that exist in Object-oriented Programming (OOP). An issue here is how to integrate the declarative nature of constraints with the step-wise execution and mutable state of OOP. To make the resulting system more usable, it should be fully backwards compatible with an existing object-oriented system, inherently object-oriented in its use of abstraction mechanisms, and have a minimal impact on the host language, as to not require learning a new, separate language to use constraints.

Solvers Are Highly Specialized

Although general constraint solving is NP-hard, most constraint solvers find good solutions in reasonable time when used in the right way for the right kinds of problems. However, to require programmers to learn the special features for each solver is undesirable. For example, some solvers including Cassowary and DeltaBlue have support for incremental resolving. Incremental resolving splits the work of the solvers into two distinct phases, and, if only a subset of the variables is known to change frequently, optimizes the solving process for cases when these change. Besides such specific features, most solvers are optimized for specific domains of problems, such as linear or non-linear equations of reals or integers or boolean logic.

In OOP, developers do not usually have to consider the lookup process itself or the way objects are laid out in memory. Similarily, Object-Constraint Programming (OCP) programmers should not in general have to deal with the choice of solver. An OCP language should, as much as possible, relieve the developer from having to answer the following questions: Which solver is the right one for my problem, i.e., which solver provides the best results or the best performance? If none of the available solvers can solve my problem, why? How might I change the constraints to allow a solver to find a solution? Can I split the constraints and hand them to different solvers which each find part of the solution? If the solvers come up with a surprising solution, which constraints allow them to do so? Thus, to solve wide

variety of problems effectively, a combination of different solvers is desirable, and, in addition, the language should aid the programmer in choosing the right solver and the correct way to invoke it to achieve optimal performance.

Practical Performance Requires Trade-Offs

A general design for an OCP language can be implemented in different ways, with different trade-offs regarding symbiosis, performance, and applicability. An implementation with virtual machine (VM) support can provide very good performance, but is not always feasible. For example, an application server can run a modified VM to make use of OCP in server code. On the other hand, code that is written in a language with multiple VM implementations and that is meant to be executed on many heterogeneous clients cannot rely on a modified VM to be available on each client. To make use of OCP in that context, it must be implemented as a user-level library. In that case, the choice of host language determines how easy syntactic extensions are and if semantic extensions can still be reasonably debugged and accessed on a meta-level.

Dynamic, object-oriented languages achieve good performance mostly due to advanced JIT techniques. Most VMs abstract from the underlying system stack and implement an abstraction for stack frames and execute everything, from local variable access to message sends, on top of this abstraction, rather than re-using the fast system stack. One feature of JIT compilers is to lower these abstractions onto the underlying system and thus achieve near-native performance for things like stack frame creation and local variable access. This works well because local variables in most OOP languages simply store pointers to objects on the heap, and thus can be directly represented by a machine word on the stack.

Variables in constraint solvers behave differently, and imperative code that interacts with libraries of constraint solvers often has to copy variables (which the JIT meticulously mapped to the stack for performance) into a solver-specific structure on the heap. The inefficiency my thus deter from using a constraint solver in the first place. Languages such as Kaleidoscopeor Turtle, which integrate the constraints directly into the language to avoid the explicit copying, have only a cosmetic effect, but cannot improve performance. To the contrary, when every variable can potentially be used in a constraint, they need more structure and are thus further removed from a machine friendly representation that the JIT can map to the stack.

cite

cite

1.2. Contributions

Babelsberg, a Language Design and Formal Semantics for Object-Constraint Programming We present a concrete design for a family of OCP languages called Babelsberg that supports a useful and expressive language for constraints, integrated with a host object-oriented language in a clean way. A heuristic for its

design is that when there is a choice, we favor simplicity over power, if that power has not proven useful in practice or would make it difficult to know what the outcome of a particular operation may be. This comes at the cost of omitting some interesting, but little used features of constraint programming, but makes the interactions between constraints and objects clearer. This heuristic also makes the Babelsberg design more independent of the particular host language and solvers used in implementing it, because only a small set of key operations have to be adapted.

For our design, we provide a formal semantics that incrementally adds OCP features to an object-oriented host language. Of particular interest here is the way we translate constraints involving methods to the solver and our rules for restricting side effects in constraint expressions. The only restrictions are: a) an expression that is used as a constraint must evaluate to a boolean (the constraint is that it evaluate to true) and b) the expression should return the same result on repeated evaluation (so that, for example, a random number generator would not qualify). Our semantics clarifies important design decision to guide language implementations, and omits details that are inherited from the underlying host language. This makes flexible enough to apply to a variety of object-oriented languages. In particular, our semantics specify: First, the interactions among constraints on identity, types, and values. Second, the addition of a novel kind of structural type-checking combined with soft constraints to tame the power of the solver with respect to changing object structure and type to satisfy constraints. Third, The addition of *value classes*, which create immutable objects for which identity is not significant. And fourth, a set of restrictions on constraints that make it easier to reason about which solutions are acceptable with respect to object identity, type, and directionality.

We also present an executable version of our semantics to automatically verify a suite of example programs that illustrate the problems we address. Our executable semantics includes framework to generate language test suites from our suite of example programs written in the executable semantics language, so concrete implementations of our design can be automatically tested for conformance to this semantics, and the test suite can be kept up-to-date as the semantics evolves.

Linguistic and Meta-Level Symbiosis with an Object-Oriented Host Language Our syntax and semantics are true extensions and thus backwards compatible with the existing object-oriented paradigm. Our syntax extensions merely integrate constraint definitions into the host language, and the semantic model for Babelsberg behaves like an ordinary object-oriented language in the absence of constraints. On the syntactic level, our additions are relatively minor, for example, to add syntactic sugar to annotate variables and expressions as read-only for the solver, or to pass various arguments to the solver if desired. Our syntactic integration ensures that constraint expressions very closely resemble assertions (with the difference that they are actually solved, rather than just tested), making it easy for the programmer to write and read constraints and object-oriented code.

Our modifications to the object-oriented semantics of the host language include dynamic typing, object encapsulation, classes and instances or prototypes with methods, and message sends. Our design supports placing constraints on the results of message sends rather than just on object attributes — thus, we argue, being more compatible with object-oriented encapsulation and abstraction than prior approaches in Constraint-Imperative Programming (CIP).

Additionally, Babelsberg defines meta-level facilities that allow libraries to construct soft as well as hard constraints, and constraints that support incremental solving. These meta-level facilities allow access to solver-specific features, provide ways to dynamically activate and deactivate constraints, and allow developers to build their own abstractions on top of the constraint extensions. Again, we believe this makes our design more compatible with the spirit of dynamic, purely object-oriented languages in which the meta-level is readily accessible.

An Architecture for Adding and Using Multiple Cooperating Solvers Babelsberg provides an architecture that supports multiple constraint solvers, which makes it straightforward to add new solvers, and which does not privilege the solvers provided with the basic implementation. To solve more complex problems involving different types of constraint, we present an architecture to let multiple solvers cooperate. Our architecture works with and without VM support and supports incremental constraint satisfaction across solvers.

Performant Implementation Techniques and Possible Applications for OCP We describe three working prototype systems. The first implements a VM-extension to support OCP, and is integrated with a state of the art Ruby virtual machine and JIT compiler. In the absence of constraints, the performance of a program written in Ruby is only modestly impacted. Two more prototypes implement the library-based design in the LivelyKernel JavaScript environment and in Squeak/Smalltalk, including additional language support to write constraints conveniently. We use these prototypes to evaluate how practical language might follow the semantics while still being practical and to explore where deviations from the semantics may be worth to explore. We also evaluate the performance of our implementations to show that OCP is practical.

Our VM-based technique for implementing object constraint languages adds a primitive to switch the interpreter between imperative evaluation, constraint construction, and constraint solving modes. The first operates the interpreter in the standard fashion, except that the instructions to load and store variables check for constraints, and if present, obtain the variable's value from the constraint solver or send the new value to the solver (which may trigger a cascade of other changes to satisfy the constraints). In constraint construction mode, the expression that defines the constraint is evaluated, not for its value, but rather to build up a network of primitive constraints that represent the constraint being added. The interpreter keeps track of dependencies in the process, so that, as needed, the solver can be activated or the code to construct the constraint can be re-

evaluated. With our technique we allow JIT compilers to optimize variables that do not participate constraints on them in the normal fashion, and thus restrict the overhead of explicitly creating variable structures on the heap to those variables that need to be manipulated by the solver.

When a VM extension with a primitive is not feasible, our library-based design includes principled restrictions that allow OCP to be implemented as user-level code without VM support. We demonstrate that our restrictions are minimal, and that practical applications can be written within these restrictions.

Finally, we present techniques for applications to interact with constraints, and report our experiences regarding the use of constraints in conjunction with existing applications, libraries, and frameworks. For interactive use, we use applications using the Morphic framework in the Squeak and LivelyKernel. We also show non-interactive, server-style applications and compare how constraint use differs between those two types of applications.

1.3. Outline

1 page

2. State of the Art

This chapter first gives an overview of the current state of the art in using constraints from imperative languages, and we elaborate on relevant theory. Our work address a number of problems which arise from previous work in the area from the dichotomy between constraints and imperative programs.

2.1. Objects and Determinism in Constraints

Object-oriented, imperative code usually mutates state explicitly over time. Any changes to objects and their structure is triggered by explicit programmer action. In contrast, constraints model a set of desirable states and and state changes arise from the solving of those constraints. The concrete changes are not necessarily deterministic: when multiple equally valid solutions are possible, different solvers may return any one or all of them. From the point of view of an object-oriented developer using constraints, this uncertainty about the selected solution may sometimes appear unintuitive when the solver picks a solution that the human programmer did not even consider.

Using constraints in conjunction with imperative code often requires programmers to consider the solving process in order to use constraints effectively; even small examples contain the potential for surprising solutions. As an example, consider a bank account application in which we want to prevent changes that would make the account balance drop below a certain threshold. Additionally, we want to track the daily interest, but not allow it to rise above 10 euros. We add a constraint to ensure this:

$$account.balance \leq minimalBalance \wedge dailyInterest \leq 10$$

We use another constraint to relate the dailyInterest to the account balance:

$$dailyInterest = (account.balance \times 0.01)/365.0$$

This second constraint can conflict with the first when the account's balance becomes large—we define it so that the second constraint can be disregarded by the solver in that case. Already, this example contains the potential for surprising solutions. The programmer expects the constraints to affect the bank's balance as well as the daily interest. However, in the absence of other restrictions, the solver is also allowed to change the minimalBalance, or set every variable to 0.

Another issue arises from the fact that in this example, we assume that the solver knows how to access the `balance` property on the account. But if another variable

with a `balance` field is known to the solver, can it change the value of the `account` variable to point to it? And what if no such field exists? Some systems, such as Kaleidoscope or Turtle, would allow the solver to invent the field or change the account variable in that case. While one might argue that such behavior is desirable in this case, it is a slippery slope. For example, consider a constraint of the form $p.x > 0 \vee p.y > 0$ on a point object `p`. There is inherent non-determinism here—the solver can choose which property to update if the constraint does not hold. But if `p` lacks both properties which one should be invented? The Kaleidoscope language, particularly the early versions, was arguably too powerful in ways like this that were interesting, but not useful in practice. This made it difficult to understand what the result of a program might be and also difficult to implement efficiently.

cite

Read-only Annotations We can remove the first case of non-determinism from the above example using the concept of read-only annotations[**borning-lisp-symbolic-computatio**]. Specifically, as `minimalBalance` is supposed to be a constant in this case, we want a way to prevent the solver from changing the value of the variable to satisfy the constraint. **borning-lisp-symbolic-computation-1992** present a declarative specification of read-only annotations, adapted from formalizations of read-only annotations in committed-choice logic languages [**maher-iclp-1987**]—we review here its intuition for reference.

Intuitively, read-only annotations in a constraint mean that information can flow out of a variable, but not into it during the solving process. This is also the case, for example, in many one-way data-flow systems, including a spreadsheet. A formula to add two values in a spreadsheet can be represented as constraints using read-only annotations. Here, the solver can only set z to be the sum of x and y:

$$\text{required} \quad x? + y? = z$$

Read-only annotations are *per constraint*, and they interact with constraint hierarchies. They are not, however, integrated into the error functions or the comparators—for example, it is not possible to ignore the annotation, even if that meant satisfying a higher-priority constraint. Consider the following constraints:

$$\begin{aligned} \text{required} \quad & x? + 5 = y \\ \text{medium} \quad & y = 20 \\ \text{low} \quad & x = 0 \end{aligned}$$

The solution is $\{x \mapsto 0, y \mapsto 5\}$—the read-only annotation on the x in the required constraint prevents the solver from satisfying the $y = 20$ constraint instead of the $x = 0$ one.

Stay Constraints *Stay constraints* can remove the surprising solutions in which the solver sets all variables to o. When using constraints in interactive applications including mutable objects, the state of which can change over time, it is important to consider such change in the solving process. In particular, we want to express to the solver that, if left undisturbed, the system state should not change over time

and if it is disturbed, the changes should be minimal. **lopez1994kaleidoscope** argue that it is important in such cases to provide the constraint equivalent of "frame axioms" to specify that parts retain their old values as the system changes. The mechanism they propose builds on the theory of constraint hierarchies which allows the solver to disregard some constraints in preference to others (we discuss this concept in more detail in Section 2.3). These "frame axioms" work by adding implicit, low-priority *stay constraints* on every variable in the system. As far as the theory of non-required constraints is concerned, a stay constraint is simply an equality constraint $v = c$ for variable v and constant c with a low priority. Operationally, c will be the value of v at the time step just before calling the solver.

Stay constraints are important in an integration of constraints into an imperative language, because a developer used to imperative programming does not expect values to change if there is not reason for them to do so—without them, we would often get counter-intuitive behavior. For example, many constraints on a quad-literal—such as that opposite sides should be parallel or that each side be perpendicular to its neighbors—can be trivially satisfied if we allow the figure to collapse to a single point. This, however, is not usually be what we expect, but a result of an under-constrained system. Stay constraints alleviate this problem by guiding the solver to find solutions that are close to the current state when the system is under-constrained.

Identity Constraints Kaleidoscope'93 introduced the concept of *identity constraints*[**lopez-ecoop-1994**], which take into account a variables identity and, in conjunction with stay constraints, allow us to specify that the system is not allowed to change the value of the `account` variable above. While this addresses this particular complication, the concept alone is insufficient to prevent non-determinism. As an example, suppose we add the following constraint to the constraints above:

$$account = account2$$

This constraint requires two variables to point to the same object. Here it is not clear whether the solution will require them both to point to the object stored in `account` or that stored in `account2`. Further, if these objects have different structures that can lead to non-deterministic behavior in subsequent constraints that access their structure. Finally, the solver can also satisfy the identity constraint by assigning both variables to yet a third object.

2.2. Uniformity for Imperative and Declarative Code

Approaches to integrating constraints with imperative code require trade-offs between the declarative and imperative programming worlds. If the constraints are expressed explicitly in imperative code, the solving may be brittle; when using a constraint solver library, objects have to be converted to certain special types to be passed to the solver; in DSLs, constraints may refer to object state directly,

violating the uniform access principle; and in CIP languages, constraints use special behavior definitions separate from methods, which violates the uniform access principle. These lead to problems as programs grow and implementations of objects change, because the constraints have to be kept up-to-date regarding the internal state of the domain objects.

Consider a graphical application that draws a window on a virtual desktop. The window is rectangular, and implemented as a pair of points `origin` and `extent`. Suppose furthermore, that the window is implemented in Ruby, and has methods to calculate the visible area and to check whether the window is on screen. The code may look as follows:

```
class Window
  attr_accessor :origin, :extent

  def visible?
    origin.x >= 0 and origin.y >= 0
  end

  def area
    extent.x * extent.y
  end
end
```

Suppose that this window is showing some important information that should remain visible on the screen and fit within the area of the window. The constraints may be that the area of the window should be ≥ 100, and the predicate method `visible?` should return true.

Imperative Solving In a standard imperative language without constraint solving, the standard approach to dealing with this is to leave it up to the programmer to find some means to maintain the constraints. This may involve determining through which code-paths the constraints may become invalidated and instrumenting those paths. The programmer may have to re-define the setters for origin and extent to check that the newly assigned points do not violate the constraints. If there are any other methods in the window that manipulate origin and extent without going through the instrumented setters, or if the points themselves are mutable, this quickly becomes cumbersome and error-prone. If the code-base is sufficiently large, a future developer may add a method that invalidates the constraints without realizing.

To satisfy constraints imperatively, we can use, for example, aspects to satisfy these constraints explicitly whenever the rectangle changes:

```
class WindowAspect < Aspect
  def ensure_constraints(method, window, status, *args)
    window.origin.x = 0 if window.origin.x <= 0
    window.origin.y = 0 if window.origin.y <= 0
    window.extent.x = 100.0 / window.extent.y if window.area <= 100
  end
end
RectAspect.new.wrap(Rectangle, :postAdvice, /(origin|extent)=/)
```

This aspect defines the method `ensure_constraints` to execute after each assignment to either `origin` or `extent` (line 8). In the method body, the x and y values of the `origin` are set to 0 if they are negative (lines 3–4) and the window's horizontal size is changed if the window's area is less than 1-00 square pixels (line 5). While this achieves the desired effect using the same object-oriented language as the rest of the system, the programmer has to ensure that all possible code-paths are covered by the aspect. Second, the original constraints are expressed in a form that requires the programmer to infer what they are designed to do. The validity of the constraints must be checked through conditional statements and branches. If there are multiple ways in which the constraints can be invalidated, nested branching ensues, which can quickly become hard to understand. Finally, it is also not trivial to tell whether the solution for the constraints is optimal. While the above constraints seems fairly trivial, there are multiple ways to satisfy the constraints. If any of the constraints were only preferential rather than strictly required, this will be even harder. In general, without a declarative specification, it is not clear if the code finds an optimal solution, or just *a* solution (for example, a window with area 200 would also satisfy the minimum area constraint if a user tries to resize the window to be smaller than 100 square pixels).

cite complexity measure paper

Constraint Libaries We may choose to use a constraint solver library to satisfy the constraints instead of using imperative code in the aspect as follows:

```
class WindowAspect < Aspect
  def ensure_constraints(method, window, status, *args)
    ctx = Z3::Context.new
    ctx << Z3::Variable.new("extent_x", window.extent.x)
    ctx << Z3::Variable.new("extent_y", window.extent.y)
    ctx << Z3::Constraint.new("extent_x * extent_y >= 100")
    # ... same for origin constraint
    ctx.solve
    window.extent.x = ctx["extent_x"]
    window.extent.y = ctx["extent_y"]
  end
end
# boilerplate code as for purely imperative approach
```

This allows us to express the constraints clearly written, however, we now need to decompose the window to create the constraints for the solver, violating encapsulation, and we have to manually copy the solution back onto our window. Programming the constraint is very different from writing ordinary code.

Domain-specific Languages For specialized domains such as user interface layouting, solvers are available as separate DSLs that describe relations between objects that can be automatically maintained by the runtime. Examples of such DSLs are CSS , the *Mac OS X* layout specification language, and the Python GUI framework *Enamel* . This approach allows programmers to specify constraints and avoid boilerplate code to trigger constraint solving and has found widespread adoption, particularily through the Mac OS X layout system. However, these ap-

cite

cite

cite

proaches require the developer to use an additional language when programming the system.

Data-Flow and Functional Reactive Programming Some languages have built-in support for data flow, which allows programmers to express unidirectional constraints between objects and their parts. Examples of such systems are ThingLab , Scratch , the LivelyKernel/Webwerkstatt , or KScript . (Of these, only ThingLab includes a planner for propagation that breaks cycles automatically. In the other systems the programmer has to break cycles explicitely.)

cite

cite

cite

cite

Such data-flow connections can observe changes to our window's `origin` and `extent`. On each change, a transformation function is executed with the current and the previous value and returns the new value for the field. Although these systems are not constraint solvers, programmers can use constraint solvers in the function in the same manner as they would in the above solution using aspects. However, the power of the solving is limited to uni-directional constraints.

Constraint-Imperative Programming From the works of Borning on integrating constraints with objects in ThingLab , the evolution of integration lead to CIP as materialized in the Kaleidoscope system and its various extensions . Related languages include Siri , Turtle , and SOUL . Our constraints can be expressed using Kaleidoscope as follows:

cite

Freeman-Benson90

Lopez94

c

c

SOUL

```
class Rectangle
  constructor area = (n: Integer)
    always: extent.x * extent.y = n
  end

  constructor visible?
    always: origin.x >= 0
    always: origin.y >= 0
  end
end

rect = Rectangle.new
always: rect.area = 100
always: rect.visible?
```

The above code states constraints clearly, and the `constructors` concept modularizes the calculated properties `visiblity` and `area` so they can be used in constraint expressions. Such constructors, called *user-defined constraints* in Kaleidoscope and Turtle, allow objects to decompose themselves into elements for interpretation by a constraint solver. However, they are a separate concept from ordinary methods, and cannot be called from imperative code and vice-versa. Constructors require multi-method dispatch semantics, while ordinary methods can only be used as constants in constraint expressions. This means that developers have to duplicate behavior definitions if an interface is needed both in constraints and imperative code (as the `area` method is, above). This, in turn, requires developers to anticipate whether certain behavior may be used in constraints or imperative code by clients. If they do not anticipate it, clients have to revert to accessing state

directly if possible. and which concept is used depends on whether an object is used in an imperative or a declarative context. This means that programmers have to consider both interfaces of an object, depending on what they want to express.

2.3. General Constraint Solving

Although constraint satisfaction in general is NP-hard, there is a large number of such libraries that restrict themselves to a particular class of problems for which they implement optimized solving strategies. It is easy to formulate a problem that is too hard for any one of such solvers. To mitigate this problem, most constraint programming systems include multiple solvers, but they usually come with a fixed set that work on specific types. Developers usually choose different strategies based on the concrete problem to get the best results or the best performance.

cite

Constraint problems can be regarded as a graph, with n-ary operations connecting variables and constraints. Solving the constraint problem can be achieved by considering variables as inputs and outputs of the operations and traversing the graph to manipulate them, bit also by considering the operations themselves and solving multiple or all operations in the graph simultaneously. For the latter, the solving process needs a built-in understanding of the operations and the types of variables involved.

One way of classifying solvers is by the *type domain* of the constraints (e.g., if they can solve for booleans, reals, finite domains of arbitrary objects, ...). Another way is to distinguish solvers that are *complete* for a particular domain (whether they can solve any constraint involving a particular class of operations in a general way) from solvers that can only select from a given set of functions (e.g., given an equation $a + b = c$, can the solver solve only for c given a and b, or can it solve for several variables at the same time). A third classification that is interesting for our work is by whether the solver supports some particular feature, such as incremental solving, constraint hierarchies, or multiple outputs.

In the rest of this section, we present and compare solvers and relevant theory that are of particular importance to or that we have used directly with Babelsberg. This is not a completely arbitrary selection, but for each of the presented solvers, a number of other solvers exist with similar features that could also have been chosen[1].

Local Propagation The simplest algorithm to solve constraints is local propagation. DeltaBlue [**freeman1989deltablue**] is a local propagation solver with support for cyclic constraints that uses user-defined functions to propagate values. Local propagation is one of the simplest techniques to solve constraints, but it is restricted in restricted in that it only handles equalities of variables in a general way. For more complex relations, local propagation solvers require the user

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

or library builder to supply propagation functions for each direction in which they wish to satisfy the relation. In contrast to general solving strategies, which suffer from increasing complexity and decreasing performance as the complexity of the domains they handle increases, local propagation solvers like DeltaBlue are especially useful for applications where a less general but fast algorithm is preferable.

Local propagation solvers model constraint systems as an acyclic directed graph of those functions that can be solved one after another. To add the constraint $a + b = 10$ to DeltaBlue, the programmer has to supply the functions $a = 10 - b$ and $b = 10 - a$. When a is then set to 20, the solver determines that the functions it should execute are $a := 20$ and then $b := 10 - a$. Since the algorithm has no cycles, this solving strategy is in linear time with respect to the number of functions. DeltaBlue supports incremental resolving by separating the generation of the acyclic graph of solving functions from their execution. To incrementally supply multiple values to the same variable, the same execution graph can be re-used, and only the initially supplied value is updated.

One restriction of local propagation solvers including DeltaBlue is that, in general, they cannot deal with constraints that have multiple outputs, as these could lead to cycles in the propagation graph. This case arises if we tried to model $a + b = c$ as a set of functions $a = 10 - b$, $b = 10 - a$, and $c = a + b$. When just one variable changes, DeltaBlue will refuse to decide whether to update b or c in response. An extension of DeltaBlue, SkyBlue [**sannella1993skyblue**], does support multiple outputs, but at the cost of instability in the linearization of the graph and resulting non-determinism in the solutions.

The Simplex Method Cassowary [**badros2001cassowary**] is an efficient solver for simultaneous linear equations using the simplex method with support for fast incremental resolving and non-required constraints. Solvers with these features are of particular interest in graphical applications where interactive performance is important, and Cassowary was recently integrated into Mac OS X [**AppleCocoaAutoLayout**] for solving layout constraints in the user interface. To achieve its performance Cassowary exploits the fact that a user manipulates only very few variables in a graphical application at the same time (fo example, dragging a window and thus manipulating its position, but not its extent) and optimizes the solving algorithm for inputs that flow from these variables.

To solve a set of linear constraints, Cassowary uses the Simplex method, in which the constraints are first brought into a form where each equation is an equality between one variable and a formula and where the variable does not occur in any other equation. To support inequalities in this scheme, Cassowary inserts a slack variable to replace the inequality with an equality. To support non-required constraints, Cassowary then adds error variables which take the difference from the optimal solution. Once the constraints are in this form, called a *tableau*, they can already be solved or deemed unsatisfiable. As an example, consider the constraint $x \leq y - 5 \leq 20$. This is transformed first into $x + s_1 = y - 5$ and $x + s_2 = 20$, and

then in simplex form becomes $x = 20 - s_2$ and $y = 20 - s_2 + s_1 + 5$. To optimize this solution, Cassowary incrementally minimizes the sum of the squares of the error and slack variables. In this case, setting both s_1 and s_2 to 0 gives the solution $x = y = 20$.

For performant, incremental resolving the assumption is that the constraints are already known, and only a limited number of variables change. This allows the solver to re-use the existing equations in simplex form and simply replace the constants with new values and re-optimize.

Relaxation Ivan Sutherland's Sketchpad system [**sutherland-ifips-1963**] used three different solving strategies — propagation of degrees of freedom, local propagation, and relaxation. The relaxation algorithm is a fast solver for constraints over reals. It uses an incremental approximation technique, and is thus well suited to situations were the system is disturbed from a state in which all constraints are satisfied and the solver is expected to find a nearby state that satisfies them again.

Constraints in Sutherland's relaxation algorithm are expressed in terms of error functions. Each constraint has error functions for each degree of freedom it removes, that is, one error function for each variable it will determine. The error functions return a real value which indicates how far constraint is from being satisfied. For example, a constraint $2x = y^2$ would have the error function $2x - y^2$. Sketchpad solves constraints by iterating over variables one by one. For each variable, the errors for all its constraints are calculated. Then the variable is modified by a small δ, and the errors are recalculated. The resulting two data points per constraint serve to approximate the constraints as linear equations. In our example, given that x is initially 5 and y is initially 3, the error is 1. We tweak x by, for example, 0.001 and find the new error as 1.002. We determine the approximate coefficient to be 2, and thus subtract from x half the error and get 4.5. Now the error is 0 and we are finished.

When variables participate in multiple constraints, the sum of the squares of the errors for each variable is used. In any case, the system iterates until it converges or stops, if it does not after a number of iterations. An issue however, is that some modification will never let the system converge, and leave it oszillating. Furthermore, Sketchpad's relaxation will generally not find a solution that minimizes the sum of the squares of the errors, which is desirable especially in graphical applications. Van Overveld describes an alternative relaxation algorithm for geometric constraints [**van199330**] that is better suited to find such solutions at the cost of making constraints harder to specify by using one displacement function per variable rather than one error function per constraint.

Backtracking BackTalk [**pachet1995**] is a finite domain constraint solving library that uses backtracking and arc-consistency optimizations to search for solutions in a finite set of objects. Finite domain solvers are very different from the aforementioned solvers, as they do not restrict themselves to a particular set of domains for efficiency. The disadvantage is that they have to use general purpose techniques

like backtracking to find solutions, which have $O(N!)$ complexity in the worst case. However, optimization techniques can improve the practical performance of such solvers to make them useful in interactive applications.

In finite domain solvers, constraints are just tests, and the solver, using backtracking, progressively assigns each variable a value from its domain until all constraints are satisfied or there are no more values left to try. The most widely used technique for optimizing this process, and the one used in BackTalk, is *arc-consistency* [**waltz1972generating**], which reduces the domains of variables before and during the enumeration of values [**nadel1988tree**, **prosser1993hybrid**]. This is done by considering each constraint separately and opportunistically removing values from the domain that do not satisfy that constraint. As domains shrink this process can be used repeatedly to converge on the set of possible solutions quickly. As an example, consider the problem of map coloring, where the task is to assign one of four colors to countries on a map such that no neighboring countries have the same color. A naïive backtracking algorithm would try to assign a color to each country in turn, and only upon finishing all assignments would it test the solution. On the other hand, when BackTalk assigns a color to a country, it immediately removes that color from the domains of the countries neighbors, before continuing the selection process. This quickly reduces the tree of choices that BackTalk has to traverse to find a valid solution.

Satisfiability Modulo Theories Z3 [**de2008z3**] is a fast and comprehensive SMT solver from Microsoft Research designed for theorem proving, and with support for a wide range of type domains, including constraints over booleans, integers, reals, and finite domains. In recent years, SMT solvers have seen significant use for program analysis, verification, and testing [**bjornernuz**]. Their popularity stems from advances in the performance of the search algorithms, and the ability to include and combine theories that are frequently used in those applications, including theories for the aforementioned primitive types, but also theories for containers such as arrays, sets, and bit-vectors. Though in many applications the main purpose of SMT solvers is to *check* the satisfiability of a logical formula, they also include the facility to generate a model, that is, a set of assignments for free variables in a formula.

Roughly, Z3 solves constraints using two core modules, a congruence engine, which determines if two formulas are equivalent and thus should be equal, and a SAT Davis-Putnam-Logemann-Loveland (DPLL(T)) [**Davis:1962:MPT:368273.368557**] algorithm. The role of the latter is to incrementally build a model for the constraints by either deducing the truth value of a formula, or, if that is not possible at any step, guessing it. The algorithm also checks if the input constraints become unsatisfiable given the current model and backtracks if that is the case. The formulas or their negations (depending on whether the DPLL(T) algorithm guessed them to be true or false) are then conjugated and passed to the respective theory solvers (based on the types in the formulas). The theory solvers determine any new facts about variables and return them to the core. For example, consider the constraint

$(a + 1 = 5) \vee (a - 1 = 5)$. The DPLL(T) core guesses that both equalities can be satisfied, and passes $a + 1 = 5 \wedge a - 1 = 5$ to the real theory solver. The real theory determines that this is unsatisfiable and returns $false$ as a new fact. The core adds this to the set of constraints, discovers that this fact makes the constraint system unsatisfiable, and backtracks to make the second equality unsatisfied. The new conjunction passed to the real solver is thus $a + 1 = 5 \wedge a - 1 \neq 5$. This has a solution, and the real solver returns it. If there are any additional constraints, the congruence engine determines if any new equalities arise (this would be the case if a is shared between different theories), then those equalities are added to the conjunctions and passed to all relevant theories, which again determine new facts based on that. The process continues until a model is found, or the core exhausts the backtracking graph.

Z3, through plugins, can be made to support a number of additional theories and model finding features, such as non-required constraints and optimization [**bjornernuz**] or strings [**zheng2013z3**]. This makes Z3 useful for a wide-variety of problems that involve multiple types with required and non-required constraints.

all types get a comparison for some set of features and use-cases, with special mention of their utilization (as libraries or DSLs) in imperative languages

comparison table

a few performance results

Constraint Hierarchies Orthogonal to the solving strategies above is the theory for trading of competing, but non-required constraints. We noted in the introduction that constraint solvers offer clear semantics when dealing with multiple competing or contradictory constraints. When such constraints are used, they are generally not all strictly required at the same time—if they are, the constraint system is simply unsatisfiable. Instead, in many application domains it is useful to introduce hierarchies of non-required or *preferential* constraints that declare desirable properties, but for which it is no error condition if the solver cannot satisfy them. For reference, we informally review the semantics for trading off hierarchies of competing constraints, and refer to Borning et.al. for a complete formal treatise[**borning-lisp-symbolic-computation-1992**].

In a system with constraint hierarchies, each constraint has a priority, with a designated highest priority that is called *required*. Highest priority constraints must always be satisfied, so in the absence of any lower priority constraints, a constraint hierarchy is solved just like any constraint system. The theory allows for an arbitrary number of different constraint priorities, but Badros et.al. note that in practice only a few priorities are used in stylized ways [**badros-tochi-2001**]. In Babelsberg, we use the priorities `required`, `high`, `medium`, and `low`.

As an example, consider the following constraints over the reals:

$$\begin{array}{rl} \text{required} & x + y = 10 \\ \text{high} & x = 8 \\ \text{low} & y = 0 \end{array}$$

Not all constraints here can be satisfied simultaneously, but according to the theory there is a single solution that best satisfies the constraints: $\{x \mapsto 8, y \mapsto 2\}$. This is the only solution satisfies both the required constraint and the high-priority constraint.

Each non-required constraint in a constraint hierarchy as an associated *error function* that is specific the kind of constraint. A simple error function will just return 0 if the constraint is unsatisfied and 1 if it is. However, for reals we can give a function that increases smoothly the further a variable is from the desired value by returning the absolute difference of the current and the desired value. For example, the error for $x = 8$ is simply $|x - 8|$, and the error for $x \leq 8$ is $x - 8$ if x is larger than 8 and 0 otherwise. Thus, in our above example, the error for the unsatisfied low priority constraint is 2.

Consider the case where two constraints at different priorities cannot be satisfied at the same time, but would have the same error if we completely satisfy one or the other:

$$\begin{array}{rl} \text{high} & x = 2 \\ \text{high} & x = -2 \\ \text{low} & x = 0 \end{array}$$

Borning et.al. [**borning-lisp-symbolic-computation-1992**] describe a number of different *comparators* for specifying the desired solution in this case, *locally-predicate-better* and *weighted-sum-better*.

Locally-predicate-better finds those solutions such that any other solution would leave a currently satisfied constraint with a priority p unsatisfied, but without satisfying a higher-priority constraint. Such a solution is called Pareto-optimal. In the above example, this excludes any solution that satisfies the low-priority constraint, because this would leave both high priority constraints unsatisfied without satisfying an even higher priority constraint. There are two locally-predicate-better solutions: $\{x \mapsto 2\}$ and $\{x \mapsto -2\}$.

Weighted-sum-better finds those solutions that minimize these squared sum of the errors, weighted according to their priority. To do so, each error functions is multiplied with weights that are chosen such that no combination of lower priority constraint can ever have a greater weighted error than any one higher priority constraint. Within the same priority, the weights can differ between constraints, however. Assuming we use the same weights to both high priority constraints above, the optimal solution to the above constraints is $\{x \mapsto 0\}$, because the absolute sum of the errors of both high priority constraints is minimal at 8. If we had a constraint of the form $x + y = 10$ with lower priority constraints $x = 0$ and $y = 0$, the weighted-sum-better comparator will find an infinite number of solutions with $\{x \in [0, 10], y \in [0, 10]\}$.

While both comparators find multiple solutions, in Object-Constraint Programming we are interested in solvers that find *a* solution to a collection of constraints—if there are multiple, the solver is permitted to select one arbitrarily. This is in contrast to logic- and constraint-logic programming, where the programmer can usually access all possible solutions.

should we provide a formal explanation or refer to the other paper?

2.4. Good Performance in Practical Applications

4 pages
- no unexpected drops in performance (steady-state performance)
- refer to performance of solvers and incremental solving
- refer to state of the art in JIT, and that JITing solvers is hard (PyPy)
- show charts comparing hand-written algo with solving

Kaleidoscope provides a declarative semantics for assignment, type declaration and subclassing. However, this declarative semantics is also used if no actual constraints are in the program. In OCP, we only use the declarative semantics for assignments of variables that are also used in constraints. This means that the performance of purely imperative code is the same as on a purely imperative VM.

Incremental Re-Satisfaction Some solvers explicitly support state changes with good performance. To that end, **freeman1989deltablue** introduce the notion of *edit constraints*. As discussed in Section 2.3, constraint solving can be regarded as creating and traversing a graph of constrained variables connected by constraint-specific operations. If only a subset of the variables change frequently, the performance of the solving can be improved by generating a graph that is optimized to be traversed by starting from these few variables. Furthermore, in graphical applications user input should often be regarded as a non-required. Consider, for example, the desire that a graphical object stay visible even if the user tries to drag it outside the visible screen area. In such cases, the graphical object should follow the user interaction until it meets the border of the visible screen area. Formally, edit constraints are again simply an equality constraint $v = c$ for variable v and constant c. Operationally, it is used to model changing an input value, for example in response to the cursor position, where in this case v and c would both hold points: v would be a point in a graphical object being moved, and c would be the cursor position. Edit constraints also have a high, but not required priority—the system will attempt to accommodate the edit action, but may be prevented from doing so. In the above example, if the graphical object would leave the visible area as a result of attempting to move it too far.

Both DeltaBlue and Cassowary treat stay and edit constraints specially, allowing very fast incremental re-satisfaction of a collection of constraints as new edit values stream into the system (and the weak stay constraints provide basic stability). For DeltaBlue, this involves pre-calculating the execution plan from the edit variables. For Cassowary, the Simplex tableau is set up so that it can be efficiently re-optimized given new values for the edit variables.
- discuss how this violates encapsulation
- present performance improvements

Part II.

Object-Constraint Programming

3. Design

An overarching design goal is that in the absence of constraints, an OCP implementation such as Babelsberg should be a standard object-oriented language. We thus focus on integrating those constraint programming features that are powerful and integrate cleanly with the underlying object structures, at the cost of omitting some possibly interesting features that have proven less relevant in practice [**freeman-benson-oopsla-1990**, **freeman-benson-phd**, **freeman-benson-iccl-1992**, **freeman-benson-ecoop-1992**].

The integration of constraints into object-oriented languages presents a number of challenges and possibly confusing issues, especially with respect to how constraints interact with mutable state, method activation, inheritance, object identity, and finite collections of objects for which identity may or may not be relevant, depending on the modeled problem. Additional confusion arises from the gap between the semantics of a solver and the solutions it produces and the limits of representing that solution in an imperative runtime. For example Cassowary is complete for linear equalities over reals, but its implementation uses float values to represent reals, so a concrete solution may suffer from round-off errors. Similarly, while Z3 supports non-linear operations over reals, many of these theories are incomplete, so a way to handle solving errors is desirable that allows the programmer to distinguish if the constraint was simply too hard for the solver, or if it is indeed unsatisfiable.

In this chapter, we approach these issues incrementally and describe our complete design in three parts: Section 3.1 describes how constraint expressions involving only objects that represent primitive types in the solver are mapped to the solver, how the solving process interacts with these objects' values, and how different solvers are combined to find a solution to problems involving more types than are supported by any one solver. In this first part we focus only on those objects for which identity is not relevant. In Section 3.2 we extend our design to include standard object-oriented structures, a heap with objects for which identity is relevant, inheritance, and polymorphism. We discuss how objects for which no solver is available can participate in constraints, and what restrictions apply to methods that are called in constraints. Finally, in Section 3.3, we add constraints over collections of objects to our design. The collection protocol provided in many languages allows for powerful abstractions, and collections can map to finite domains, which are useful in a variety of constraint problems. In this third part, we describe what types of operations on collections we support in constraints in a generic manner that is independent of the concrete collection protocol and implementation of any one language.

3.1. Constraints on Primitive Types

We describe the first set of design issues using a simple language that has only primitive types, namely booleans, integers, reals, and strings. In this first step, we omit user-defined, structured classes, and assume the language has only classes that map to our primitive types, i.e., `Boolean`, `Integer`, `Float`, and `String`. The basic language we use here is object-oriented, with standard imperative evaluation rules. The language includes methods, mutable variables, and standard imperative control structures such as branches and loops, and operations on the variables are executed as method sends (with possibly some primitive behavior).

3.1.1. Declaring Constraints

The first extension over the basic object-oriented language adds support for declaring constraints. A statement starting with either the keyword `once` or `always` declares a constraint. Constraints are immediately activated and solved. A `once` constraint removed immediately after the solver finds a solution to the current set of constraints, whereas an `always` constraint remains in effect for the rest of the execution. When a constraint expression is encountered, the interpreter constructs a closure over the expression and adds it to the *constraint store*. Each time the current set of constraints must be solved, all expressions contained in this store are translated into a form that can be passed to the solver, the solver is called, and its results are translated back.

Constraint expressions are interpreted using an additional evaluation mode we call *constraint construction mode*. In constraint construction mode, methods are translated into the equivalent operations in the solver and objects are unboxed[1]. For example, 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, since these are full objects, the language may include methods that have no equivalent primitive operation in any solver. We ignore these for now, and discuss how such methods are handled in Section 3.2.

3.1.2. Translating and Solving Constraints

TODO

3.1.3. Constraints and Mutable State

The second extension to the standard imperative model and the second opportunity at which constraints must be solved are assignments. Assignments are evaluated by evaluating the right hand side as an ordinary expression. We then add a constraint to the store that equates the variable on the left hand side to the

[1] This may not be required in a language that has true primitive values like Java

evaluated result of the right hand side, and turn all constraints over to the solver. If they can be solved, the variables are updated and the temporary assignment constraint is removed from the store. Doing this ensures that assignment interacts correctly with other constraints, and that no constraints can be violated with assignment.

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).

4 pages
- these map to solver types like reals, integers, booleans, or strings
- different solvers with different levels of support for operations available
- solvers support a subset of the available operations of the host language
- solvers mustn't be allowed to change object structure
- trees of objects
- what should this do: always e.p.x == 17 ; e.p = 1@2
- mutable objects -> we must pass all constraints to the solver all the time

3.2. Constraints on Messages

7 pages
- optimization: only re-transform the constraints for objects that have changed
- lookup must work => identities must be fixed
- we need our transformation to be sound, but not necessarily complete

3.3. Constraints on Collections of Objects

4 pages
- collection apis
- sets are a nice declarative feature
- objects usually don't care what they are in
- need to worry about stuff like early return optimizations
- collections implemented on top of each other, but breaks down to complex objects problem

4. Semantics

Our formal semantics provides a complete semantics of the Babelsberg design that can be used to guide practical implementations . It is meant to as simple as possible, while still encompassing the major aspects of OCP and the important design decisions in its Babelsberg form. Because Babelsberg integrates constraints in an existing object-constraint host language, the semantics omits constructs such as exception handling for constraint solver failures, class and method definitions, and syntactic sugar, that are intended to be inherited from the host language. Our semantics instead focuses on the expression of standard object-oriented constructs that need to be modified to support the Babelsberg design. We present the semantics in the same increments as we did the design: Section 4.1 presents the semantics of a simple imperative language that has only primitive reals, integers, and booleans. Section 4.2 adds the rules for method lookup and dispatch, identity constraints, and the interaction with the heap. Section 4.3 presents the rules for constraints over collections.

and have been published in

In addition to our semantic rules, we also present an executable form of the complete semantics that passes a language test suite in Section 4.4. We use this executable form to demonstrate the validity of our rules, and provide a mechanism to generate test suites that can be run against Babelsberg implementations to assert their compliance to the semantics.

4.1. Primitive Types

We start with a very basic language, Babelsberg/PrimitiveTypes that has only primitive values. The boolean type is required to make meaningful Babelsberg programs, since by definition constraint expressions must return either true or false (with the solvers task to make them true). For this initial language, we also add reals and integers.

4.1.1. Formalism

We present the formal semantics of Babelsberg/PrimitiveTypes.

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

4.1.1.2. 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 $\mathtt{E}$ ranges over environments. When convenient we also view an environment as a set of (program variable, program value) pairs. For each operator $\mathtt{o}$ in the language we assume the existence of a corresponding semantic function denoted $[\![\mathtt{o}]\!]$.

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

"Expression $\mathtt{e}$ evaluates to value $\mathtt{v}$ in the context of environment $\mathtt{E}$."

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

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

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

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

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

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

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

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

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

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

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

$\boxed{\mathtt{<E|C|s>} \rightarrow \mathtt{<E'|C'>}}$

"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.

$$\frac{\mathtt{E} \vdash \mathtt{e} \Downarrow \mathtt{v} \qquad \mathsf{stay}(\mathtt{E}) = \mathtt{C}_s \qquad \mathtt{E}' \mid (\mathtt{C} \wedge \mathtt{C}_s \wedge \mathtt{x} = \mathtt{v})}{\mathtt{<E|C|x := e>} \rightarrow \mathtt{<E'|C>}} \text{ (S-Asgn)}$$

$$\frac{\mathsf{stay}(\mathtt{E}) = \mathtt{C}_s \qquad \mathtt{E}' \mid (\mathtt{C} \wedge \mathtt{C}_s \wedge \mathtt{C}_0)}{\mathtt{<E|C|once\ C_0>} \rightarrow \mathtt{<E'|C>}} \text{ (S-Once)}$$

$$\frac{\mathtt{<E|C|once\ C_0>} \rightarrow \mathtt{<E'|C>} \qquad \mathtt{C}' = \mathtt{C} \wedge \mathtt{C}_0}{\mathtt{<E|C|always\ C_0>} \rightarrow \mathtt{<E'|C'>}} \text{ (S-Always)}$$

$$\mathtt{<E|C|skip>} \rightarrow \mathtt{<E|C>} \text{ (S-Skip)}$$

$$\frac{\mathtt{<E|C|s_1>} \rightarrow \mathtt{<E'|C'>} \qquad \mathtt{<E'|C'|s_2>} \rightarrow \mathtt{<E''|C''>}}{\mathtt{<E|C|s_1;s_2>} \rightarrow \mathtt{<E''|C''>}} \text{ (S-Seq)}$$

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

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

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

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

4.2. Objects and Messages

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.

```
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.

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

This program fails the structural compatibility checks as well:

```
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).

For our semantics we use a simple language, Babelsberg/Objects, that includes primitive types (reals, booleans, and strings), as well as mutable objects that live on the heap and immutable objects on the stack. Its semantic rules are mostly standard imperative rules, with the addition of rules to assert and maintain constraints. The semantics for this language, although not object-oriented, illustrates our key contributions. Extending it to include all the essential features of the actual object-constraint languages, such as support for classes, methods, messages, and inheritance, along with constraint definitions that can include method calls, is straightforward, and is presented in a companion technical report [**felgentreff:2014:semantics**].

In addition to the imperative semantics, after every statement execution the current set of constraints is solved and the environment and heap are replaced with the solution. The current set of constraints is determined as follows. For assignments, we evaluate the right-hand side and then create a constraint that the resulting value and the left-hand side be identical. Otherwise the statement is just executed. In addition to any constraint resulting from an assignment statement, each constraint in the current set of constraints (e.g, a constraint from an explicit `always` statement, or from an implicit weak stay) is given to the solver. If the solver finds a solution, the environment and heap are replaced by the model the solver found; if the constraints have no solution (or they are too hard), the execution halts. In practical languages, a run-time exception would be generated and the heap and environment remain unchanged.

The semantics treats the solver as a black box, but we assume it supports our primitive types, records, uninterpreted functions, as well as hard and soft constraints [**borning1992constraint**] and read-only variables. The solver should find a single best solution — if there are multiple solutions, the solver is free to pick any one of them.

4.2.1. 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.

We use the following syntax for Babelsberg/Objects:

Statement	s	::=	`skip` | `L := e` | `L := 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` | `e` $\oslash$ `e` | `e` $\otimes$ `e` | `I` | `o` | `new o` | `e.l(`e_1`,...,`e_n`)` | `D`
Identity	I	::=	`L == L`
Object Literal	o	::=	`{`l_1`:`e_1`,...,`l_n`:`e_n`}`
L-Value	L	::=	`x` | `L.l`
Constant	c	::=	`true` | `false` | `nil` | base type constants
Variable	x	::=	variable names
Label	l	::=	record label names
Reference	r	::=	references to heap records
Dereference	D	::=	$\mathbb{H}$`(e)`
Method Body	b	::=	`s; return e` | `return e`
Value	v	::=	`c` | `r` | `{`l_1`:`v_1`,...,`l_n`:`v_n`}`

Metavariable `c` ranges over the `nil` value, booleans, and primitive type constants. A finite set of arithmetic operators on expressions is ranged over by $\oplus$, $\oslash$ ranges over a set of *predicate* operators, and $\otimes$ ranges over logical operators for combining

boolean expressions. $\oslash$ includes at least an equality operator = for each primitive type, and $\otimes$ includes at least conjunction $\wedge$. The operator == tests for identity — for primitive values and immutable objects this is the same as =. The symbol ρ ranges over constraint *priorities* and is assumed to include a bottom element *weak* and a top element *required*. The syntax requires the priority to be explicit; for simplicity we sometimes omit it in the rules and assume a priority of *required*.

Finally, in the syntax, we treat $\mathbb{H}$as a primitive operation for dereferencing. Source programs will not use expressions of the form $\mathbb{H}$`(e)`, but they are introduced as part of constraints given to the solver, which we assume will treat $\mathbb{H}$as an uninterpreted function.

4.2.2. Operational Semantics

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

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

"Expression e evaluates to value v in the context of environment $\mathbb{E}$ and heap $\mathbb{H}$."

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

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

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

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{L} \Downarrow \mathsf{v} \qquad \mathsf{v} = \{\mathsf{l}_1:\mathsf{v}_1,\ldots,\mathsf{l}_n:\mathsf{v}_n\} \qquad 1 \le i \le n}{\mathbb{E};\mathbb{H} \vdash \mathsf{L}.\mathsf{l}_i \Downarrow \mathsf{v}_i} \qquad \text{(E-ValueField)}$$

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e}_1 \Downarrow \mathsf{v}_1 \cdots \mathbb{E};\mathbb{H} \vdash \mathsf{e}_n \Downarrow \mathsf{v}_n}{\mathbb{E};\mathbb{H} \vdash \{\mathsf{l}_1:\mathsf{e}_1,\ldots,\mathsf{l}_n:\mathsf{e}_n\} \Downarrow \{\mathsf{l}_1:\mathsf{v}_1,\ldots,\mathsf{l}_n:\mathsf{v}_n\}} \qquad \text{(E-Object)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E};\mathbb{H} \vdash e_2 \Downarrow v_2 \qquad v_1 \ [\![\oplus]\!] \ v_2 = v}{\mathbb{E};\mathbb{H} \vdash e_1 \ \oplus \ e_2 \Downarrow v} \quad \text{(E-Op)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E};\mathbb{H} \vdash e_2 \Downarrow v_2 \qquad v_1 \ [\![\ominus]\!] \ v_2 = v}{\mathbb{E};\mathbb{H} \vdash e_1 \ \ominus \ e_2 \Downarrow v} \quad \text{(E-Compare)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E};\mathbb{H} \vdash e_2 \Downarrow v_2 \qquad v_1 \ [\![\otimes]\!] \ v_2 = v}{\mathbb{E};\mathbb{H} \vdash e_1 \ \otimes \ e_2 \Downarrow v} \quad \text{(E-Combine)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash L_1 \Downarrow v \qquad \mathbb{E};\mathbb{H} \vdash L_2 \Downarrow v}{\mathbb{E};\mathbb{H} \vdash L_1 \texttt{ == } L_2 \Downarrow \texttt{true}} \quad \text{(E-IdentityTrue)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash L_1 \Downarrow v_1 \qquad \mathbb{E};\mathbb{H} \vdash L_2 \Downarrow v_2 \qquad v_1 \neq v_2}{\mathbb{E};\mathbb{H} \vdash L_1 \texttt{ == } L_2 \Downarrow \texttt{false}} \quad \text{(E-IdentityFalse)}$$

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

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

"Expression e has type T in the context of environment $\mathbb{E}$ and heap $\mathbb{H}$."

"Constraint C is well formed in the context of environment $\mathbb{E}$ and heap $\mathbb{H}$."

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, in support of the goals listed in **??**. We formalize our notion of structure through a simple syntax of types:

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

The typechecking rules are mostly standard. We check expressions dynamically just before constraint solving, so we typecheck in the context of a run-time 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.

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

$$\frac{\mathbb{E};\mathbb{H} \vdash o : \{l_1 : T_1, \ldots, l_n : T_n\} \qquad \mathbb{H}(r)=o}{\mathbb{E};\mathbb{H} \vdash r : \{l_1 : T_1, \ldots, l_n : T_n\}} \quad \text{(T-Ref)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash e : \{l_1 : T_1, \ldots, l_n : T_n\}}{\mathbb{E};\mathbb{H} \vdash \mathbb{H}(e) : \{l_1 : T_1, \ldots, l_n : T_n\}} \quad \text{(T-Deref)}$$

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

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

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

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e}_1 : \mathsf{T} \qquad \mathbb{E};\mathbb{H} \vdash \mathsf{e}_2 : \mathsf{T}}{\mathbb{E};\mathbb{H} \vdash \mathsf{e}_1 \oslash \mathsf{e}_2 : \mathtt{PrimitiveType}} \quad \text{(T-Compare)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e}_1 : \mathtt{PrimitiveType} \qquad \mathbb{E};\mathbb{H} \vdash \mathsf{e}_2 : \mathtt{PrimitiveType}}{\mathbb{E};\mathbb{H} \vdash \mathsf{e}_1 \otimes \mathsf{e}_2 : \mathtt{PrimitiveType}} \quad \text{(T-Combine)}$$

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

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

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

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

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

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

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

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

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

The semantics ensure that each variable has a weak stay constraint to keep it at its current value, if possible. To ensure that the solver cannot invent new fields on objects, we create required stays on the heap that assert the structure of the objects. The first two stay rules assert only weak stays on all variables and are used for assignment, which can change object identities. The i-stay($\mathbb{E}$) = C and i-stay($\mathbb{H}$) = C rules use the i-stay(e) = C rule to ensure that object identities and structures cannot be affected when solving value constraints — they create weak

stay constraints on primitive values, but require references and the structure of objects to remain unchanged.

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

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

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

$$\frac{\mathbb{H}(\texttt{r}) = \texttt{o} \qquad \mathbb{H}_0 = \mathbb{H} \backslash \{(\texttt{r}, \texttt{o})\} \qquad \text{stay}(\mathbb{H}_0) = \texttt{C}_0 \qquad \texttt{o} = \{\texttt{l}_0\texttt{:v}_0, \ldots, \texttt{l}_n\texttt{:v}_n\} \qquad \texttt{C} = \texttt{weak x}_{r0}\texttt{ = v}_0 \wedge \ldots \wedge \texttt{weak x}_{rn}\texttt{ = v}_n}{\text{stay}(\mathbb{H}) = \texttt{C}_0 \wedge \texttt{weak } \mathbb{H}\texttt{(r)=\{l}_0\texttt{:x}_{r0}, \ldots, \texttt{l}_n\texttt{:x}_{rn}\texttt{\}} \wedge \texttt{C}} \qquad \text{(StayHeap)}$$

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

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

$$\frac{\text{i-stay}(\texttt{x}_{x0}\texttt{=v}_0) = \texttt{C}_{x0} \cdots \text{i-stay}(\texttt{x}_{xn}\texttt{=v}_n) = \texttt{C}_{xn} \qquad \texttt{C}_x = \texttt{C}_{x0} \wedge \ldots \wedge \texttt{C}_{xn}}{\text{i-stay}(\texttt{x=\{l}_0\texttt{:v}_0, \ldots, \texttt{l}_n\texttt{:v}_n\texttt{\}}) = \texttt{ required x=\{l}_0\texttt{:x}_{x0}, \ldots, \texttt{l}_n\texttt{:x}_{xn}\texttt{\}} \wedge \texttt{C}_x} \qquad \text{(IStayObject)}$$

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

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

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

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

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

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 $\mathbb{H}$`(L).l` (recursively, as required). We do not give rules, because they are straightforward. The full Babelsberg/Objects language includes a similar judgment that also inlines method calls, given in its entirety in the companion technical report.

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

"Execution starting from configuration <$\mathbb{E}$|$\mathbb{H}$|$\mathbb{C}$|$\mathbb{I}$|s> ends in the state <E′|$\mathbb{H}'$|C′|$\mathbb{I}'$>."

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 first two rules below describe 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 these rules get stuck if 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.

$$\frac{\begin{array}{c}\mathbb{E};\mathbb{H} \vdash \texttt{e} \Downarrow \texttt{v} \qquad \text{stay}(\mathbb{E}) = \texttt{C}_{E_s} \qquad \text{stay}(\mathbb{H}) = \texttt{C}_{H_s} \\ \texttt{L} \neq \texttt{x} \Rightarrow \mathbb{E};\mathbb{H} \vdash \texttt{L} : \texttt{T} \qquad \mathbb{E};\mathbb{H} \vdash \mathbb{I} \wedge \texttt{L=v} \rightsquigarrow \texttt{C}_I \qquad \mathbb{E}';\mathbb{H}' \models (\texttt{C}_I \wedge \texttt{C}_{E_s} \wedge \texttt{C}_{H_s}) \\ \text{i-stay}(\mathbb{E}') = \texttt{C}_{E'_s} \qquad \text{i-stay}(\mathbb{H}') = \texttt{C}_{H'_s} \\ \mathbb{E};\mathbb{H} \vdash \mathbb{C} \wedge \texttt{L=v} \rightsquigarrow \texttt{C}_C \qquad \mathbb{E}';\mathbb{H}' \vdash \texttt{C}_C \qquad \mathbb{E}'';\mathbb{H}'' \models (\texttt{C}_C \wedge \texttt{C}_{E'_s} \wedge \texttt{C}_{H'_s})\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|L := e>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{>}} \text{(S-Asgn)}$$

$$\frac{\begin{array}{c}\mathbb{E};\mathbb{H} \vdash \texttt{e}_1 \Downarrow \texttt{v}_n \cdots \mathbb{E};\mathbb{H} \vdash \texttt{e}_n \Downarrow \texttt{v}_n \\ \mathbb{H}(\texttt{r})\uparrow \qquad \mathbb{H}_0 = \mathbb{H} \bigcup \{(\texttt{r}, \{\texttt{l}_1\texttt{:v}_1, \ldots, \texttt{l}_n\texttt{:v}_n\})\} \\ \text{stay}(\mathbb{E}) = \texttt{C}_{E_s} \qquad \text{stay}(\mathbb{H}_0) = \texttt{C}_{H_s} \qquad \texttt{C}_{new} = \texttt{L=r} \wedge \mathbb{H}\texttt{(r)=}\{\texttt{l}_1\texttt{:v}_1, \ldots, \texttt{l}_n\texttt{:v}_n\} \\ \texttt{L} \neq \texttt{x} \Rightarrow \mathbb{E};\mathbb{H} \vdash \texttt{L} : \texttt{T} \qquad \mathbb{E};\mathbb{H}_0 \vdash \mathbb{I} \wedge \texttt{C}_{new} \rightsquigarrow \texttt{C}_I \qquad \mathbb{E}';\mathbb{H}' \models (\texttt{C}_I \wedge \texttt{C}_{E_s} \wedge \texttt{C}_{H_s}) \\ \text{i-stay}(\mathbb{E}') = \texttt{C}_{E'_s} \qquad \text{i-stay}(\mathbb{H}') = \texttt{C}_{H'_s} \\ \mathbb{E}';\mathbb{H}' \vdash \mathbb{C} \wedge \texttt{C}_{new} \rightsquigarrow \texttt{C}_C \qquad \mathbb{E}';\mathbb{H}' \vdash \texttt{C}_C \qquad \mathbb{E}'';\mathbb{H}'' \models (\texttt{C}_C \wedge \texttt{C}_{E'_s} \wedge \texttt{C}_{H'_s})\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|L := new } \{\texttt{l}_1\texttt{:e}_1, \ldots, \texttt{l}_n\texttt{:e}_n\}\texttt{>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{>}} \text{(S-AsgnNew)}$$

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{L}_0 \Downarrow \mathsf{v} \qquad \mathbb{E};\mathbb{H} \vdash \mathsf{L}_1 \Downarrow \mathsf{v}}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{once } \mathsf{L}_0 \texttt{ == } \mathsf{L}_1 \rangle \rightarrow \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} \rangle} \text{ (S-OnceIdentity)}$$

$$\frac{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{once } \mathsf{L}_0 \texttt{ == } \mathsf{L}_1 \rangle \rightarrow \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} \rangle \qquad \mathbb{I}' = \mathbb{I} \wedge \mathsf{L}_0 = \mathsf{L}_1}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{always } \mathsf{L}_0 \texttt{ == } \mathsf{L}_1 \rangle \rightarrow \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I}' \rangle} \text{ (S-AlwaysIdentity)}$$

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.

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{C}_0 \qquad \text{i-stay}(\mathbb{E}) = \mathsf{C}_{E_s} \qquad \text{i-stay}(\mathbb{H}) = \mathsf{C}_{H_s} \qquad \mathbb{E};\mathbb{H} \vdash \mathbb{C} \wedge \mathsf{C}_0 \rightsquigarrow \mathsf{C}' \qquad \mathbb{E}';\mathbb{H}' \models (\mathsf{C}' \wedge \mathsf{C}_{E_s} \wedge \mathsf{C}_{H_s})}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{once } \mathsf{C}_0 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C} | \mathbb{I} \rangle} \text{ (S-Once)}$$

$$\frac{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{once } \mathsf{C}_0 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C} | \mathbb{I} \rangle \qquad \mathbb{C}' = \mathbb{C} \wedge \mathsf{C}_0}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{always } \mathsf{C}_0 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I} \rangle} \text{ (S-Always)}$$

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

$$\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{skip} \rangle \rightarrow \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} \rangle \text{ (S-Skip)}$$

$$\frac{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \mathsf{s}_1 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle \qquad \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' | \mathsf{s}_2 \rangle \rightarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' \rangle}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \mathsf{s}_1 \texttt{;} \mathsf{s}_2 \rangle \rightarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' \rangle} \text{ (S-Seq)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e} \Downarrow \texttt{true} \qquad \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \mathsf{s}_1 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{if e then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle} \text{ (S-IfThen)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e} \Downarrow \texttt{false} \qquad \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \mathsf{s}_2 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{if e then } \mathsf{s}_1 \texttt{ else } \mathsf{s}_2 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle} \text{ (S-IfElse)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e} \Downarrow \texttt{true} \qquad \langle \mathbb{E} | \mathbb{C} | \mathbb{H} | \mathbb{I} | \mathsf{s} \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle \qquad \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' | \texttt{while e do s} \rangle \rightarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' \rangle}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{while e do s} \rangle \rightarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' \rangle} \text{ (S-WhileDo)}$$

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathsf{e} \Downarrow \texttt{false}}{\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{while e do s} \rangle \rightarrow \langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} \rangle} \text{ (S-WhileSkip)}$$

4.2.3. Key Properties

Here we state two key theorems about our formalism. The proofs for these theorems can be found in the companion technical report. 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* $\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{(once|always)}\ C_0 \rangle \rightarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' \rangle$ *and* $\mathbb{E}; \mathbb{H} \vdash C_0$ *and* $\mathbb{E}; \mathbb{H} \vdash x : T$*, then* $\mathbb{E}'; \mathbb{H}' \vdash x : T$*.*

The second theorem formalizes the idea that all solutions to an assignment will produce structurally equivalent environments and heaps (provided we start in a well-formed configuration where all identity constraints are satisfied):

Theorem 2. *(Structural Determinism) If* $\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | L := e \rangle \rightarrow \langle \mathbb{E}_1 | \mathbb{H}_1 | \mathbb{C}_1 | \mathbb{I}_1 \rangle$ *and* $\langle \mathbb{E} | \mathbb{H} | \mathbb{C} | \mathbb{I} | L := e \rangle \rightarrow \langle \mathbb{E}_2 | \mathbb{H}_2 | \mathbb{C}_2 | \mathbb{I}_2 \rangle$ *and* $\mathbb{E}; \mathbb{H} \vdash I \Downarrow \texttt{true}$ *and* x *in dom*$(\mathbb{E})$*, then* $\mathbb{E}_1; \mathbb{H}_1 \vdash x : T$ *and* $\mathbb{E}_2; \mathbb{H}_2 \vdash x : T$*.*

Since we have methods with local scopes now, we introduce local environments S as mappings from local names to globally unique names in $\mathbb{E}$. 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 now store constraints in pairs with the local environment they were created in.

The rest of this appendix just gives the judgments and rules for Babelsberg/Objects. A technical report is available with a more complete discussion .

CITE

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

"Lookup of method $\mathtt{l}$ in the object or value v returns the formal parameter names x_1 through x_n and the method body b"

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

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

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

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

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

$$\boxed{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}\rangle}$$

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

$$\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{c}\rangle \Downarrow \langle \mathbb{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{c}\rangle \quad (\text{E-Const})$$

$$\frac{\mathsf{S}(\mathsf{x}) = \mathsf{x}_g \qquad \mathbb{E}(\mathsf{x}_g) = \mathsf{v}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{x}\rangle \Downarrow \langle \mathbb{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{v}\rangle}\ (\text{E-Var})$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{r}\rangle \\ \mathbb{H}'(\mathsf{r}) = \{\mathsf{l}_1:\mathsf{v}_1,\ldots,\mathsf{l}_n:\mathsf{v}_n\} \qquad 1 \leq i \leq n\end{array}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}.\mathsf{l}_i\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}_i\rangle}\ (\text{E-Field})$$

$$\frac{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\{\mathsf{l}_1:\mathsf{v}_1,\ldots,\mathsf{l}_n:\mathsf{v}_n\}\rangle \qquad 1 \leq i \leq n}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}.\mathsf{l}_i\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}_i\rangle}\ (\text{E-ValueField})$$

$$\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{r}\rangle \Downarrow \langle \mathbb{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{r}\rangle \quad (\text{E-Ref})$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\rangle \Downarrow \langle \mathbb{E}_0|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{v}_1\rangle \\ \langle \mathbb{E}_0|\mathsf{S}|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}_2\rangle \qquad \mathsf{v}_1\ [\![\oplus]\!]\ \mathsf{v}_2 = \mathsf{v}\end{array}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\ \oplus\ \mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}\rangle}\ (\text{E-Op})$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\rangle \Downarrow \langle \mathbb{E}_0|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{v}_1\rangle \\ \langle \mathbb{E}_0|\mathsf{S}|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}_2\rangle \qquad \mathsf{v}_1\ [\![\circledcirc]\!]\ \mathsf{v}_2 = \mathsf{v}\end{array}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\ \circledcirc\ \mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}\rangle}\ (\text{E-Compare})$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\rangle \Downarrow \langle \mathbb{E}_0|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{v}_1\rangle \\ \langle \mathbb{E}_0|\mathsf{S}|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}_2\rangle \qquad \mathsf{v}_1\ [\![\circledcirc]\!]\ \mathsf{v}_2 = \mathsf{v}\end{array}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\ \circledcirc\ \mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}\rangle}\ (\text{E-Combine})$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\rangle \Downarrow \langle \mathbb{E}_0|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{v}\rangle \\ \langle \mathbb{E}|\mathsf{S}|\mathbb{H}_0|\mathbb{C}_0|\mathbb{I}_0|\mathsf{e}_2\rangle \Downarrow \langle \mathsf{S}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathsf{v}\rangle\end{array}}{\langle \mathbb{E}|\mathsf{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathsf{e}_1\ \texttt{==}\ \mathsf{e}_2\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\texttt{true}\rangle}\ (\text{E-IdentityTrue})$$

$$\frac{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e}_1 \rangle \Downarrow \langle \mathbb{E}_0 \mid \mathbb{H}_0 \mid \mathbb{C}_0 \mid \mathbb{I}_0 \mid \mathtt{v}_1 \rangle \quad \langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H}_0 \mid \mathbb{C}_0 \mid \mathbb{I}_0 \mid \mathtt{e}_2 \rangle \Downarrow \langle \mathtt{S}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \mid \mathtt{v}_2 \rangle \quad \mathtt{v}_1 \neq \mathtt{v}_2}{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e}_1 \texttt{ == } \mathtt{e}_2 \rangle \Downarrow \langle \mathbb{E}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \mid \texttt{false} \rangle} \text{ (E-IdentityFalse)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e} \rangle \Downarrow \langle \mathbb{E}_0 \mid \mathbb{H}_0 \mid \mathbb{C}_0 \mid \mathbb{I}_0 \mid \mathtt{v} \rangle \\ \mathit{lookup}(\mathtt{v},\mathtt{l}) = (\mathtt{x}_1 \cdots \mathtt{x}_n, \mathtt{s};\ \texttt{return e}) \\ \mathit{enter}(\mathbb{E}_0,\mathtt{S},\mathbb{H}_0,\mathbb{C}_0,\ \mathbb{I}_0,\mathtt{v},\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{e}_1 \cdots \mathtt{e}_n) = (\mathbb{E}_1,\mathtt{S}_m,\mathbb{H}_1,\mathbb{C}_1,\mathbb{I}_1) \\ \langle \mathbb{E}_1 \mid \mathtt{S}_m \mid \mathbb{H}_1 \mid \mathbb{C}_1 \mid \mathbb{I}_1 \mid \mathtt{s} \rangle \rightarrow \langle \mathbb{E}' \mid \mathtt{S}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \rangle \\ \langle \mathbb{E}' \mid \mathtt{S}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \mid \mathtt{e} \rangle \Downarrow \langle \mathbb{E}'' \mid \mathbb{H}'' \mid \mathbb{C}'' \mid \mathbb{I}'' \mid \mathtt{v}_r \rangle\end{array}}{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e.l}(\mathtt{e}_1,\ldots,\mathtt{e}_n) \rangle \Downarrow \langle \mathbb{E}'' \mid \mathbb{H}'' \mid \mathbb{C}'' \mid \mathbb{I}'' \mid \mathtt{v}_r \rangle} \text{ (E-Call)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e} \rangle \Downarrow \langle \mathbb{E}_0 \mid \mathbb{H}_0 \mid \mathbb{C}_0 \mid \mathbb{I}_0 \mid \mathtt{v} \rangle \\ \mathit{lookup}(\mathtt{v},\mathtt{l}) = (\mathtt{x}_1 \cdots \mathtt{x}_n, \texttt{return e}) \\ \mathit{enter}(\mathbb{E}_0,\mathtt{S},\mathbb{H}_0,\mathbb{C}_0,\ \mathbb{I}_0,\mathtt{v},\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{e}_1 \cdots \mathtt{e}_n) = (\mathbb{E}_1,\mathtt{S}_m,\mathbb{H}_1,\mathbb{C}_1,\mathbb{I}_1) \\ \langle \mathbb{E}_1 \mid \mathtt{S}_m \mid \mathbb{H}_1 \mid \mathbb{C}_1 \mid \mathbb{I}_1 \mid \mathtt{e} \rangle \Downarrow \langle \mathbb{E}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \mid \mathtt{v}_r \rangle\end{array}}{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e.l}(\mathtt{e}_1,\ldots,\mathtt{e}_n) \rangle \Downarrow \langle \mathbb{E}' \mid \mathbb{H}' \mid \mathbb{C}' \mid \mathbb{I}' \mid \mathtt{v}_r \rangle} \text{ (E-CallSimple)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e}_1 \rangle \Downarrow \langle \mathbb{E}_1 \mid \mathbb{H}_1 \mid \mathbb{C}_1 \mid \mathbb{I}_1 \mid \mathtt{v}_1 \rangle \\ \cdots \\ \langle \mathbb{E}_{n-1} \mid \mathtt{S} \mid \mathbb{H}_{n-1} \mid \mathbb{C}_{n-1} \mid \mathbb{I}_{n-1} \mid \mathtt{e}_n \rangle \Downarrow \langle \mathbb{E}_n \mid \mathbb{H}_n \mid \mathbb{C}_n \mid \mathbb{I}_n \mid \mathtt{v}_n \rangle \\ \mathbb{H}_n(\mathtt{r})\uparrow \qquad \mathbb{H}' = (\mathbb{H}_n \bigcup \{(\mathtt{r}, \{\mathtt{l}_1 : \mathtt{v}_1, \ldots, \mathtt{l}_n : \mathtt{v}_n\})\})\end{array}}{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \texttt{new } \{\mathtt{l}_1 : \mathtt{e}_1, \ldots, \mathtt{l}_n : \mathtt{e}_n\} \rangle \Downarrow \langle \mathbb{E}_n \mid \mathbb{H}' \mid \mathbb{C}_n \mid \mathbb{I}_n \mid \mathtt{r} \rangle} \text{ (E-New)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \mathtt{e}_1 \rangle \Downarrow \langle \mathbb{E}_1 \mid \mathbb{H}_1 \mid \mathbb{C}_1 \mid \mathbb{I}_1 \mid \mathtt{v}_1 \rangle \\ \cdots \\ \langle \mathbb{E}_{n-1} \mid \mathtt{S} \mid \mathbb{H}_{n-1} \mid \mathbb{C}_{n-1} \mid \mathbb{I}_{n-1} \mid \mathtt{e}_n \rangle \Downarrow \langle \mathbb{E}_n \mid \mathbb{H}_n \mid \mathbb{C}_n \mid \mathbb{I}_n \mid \mathtt{v}_n \rangle\end{array}}{\langle \mathbb{E} \mid \mathtt{S} \mid \mathbb{H} \mid \mathbb{C} \mid \mathbb{I} \mid \{\mathtt{l}_1 : \mathtt{e}_1, \ldots, \mathtt{l}_n : \mathtt{e}_n\} \rangle \Downarrow \langle \mathbb{E}_n \mid \mathbb{H}_n \mid \mathbb{C}_n \mid \mathbb{I}_n \mid \{\mathtt{l}_1 : \mathtt{v}_1, \ldots, \mathtt{l}_n : \mathtt{v}_n\} \rangle} \text{ (E-Value)}$$

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

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

The typechecking rules are unchanged from Babelsberg/UID. Note that this means that 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. New (non-value) object construction also does not typecheck, since constraints must be side-effect-free. Finally, the typechecking rules continue to only type against the global environment, they do not use the local variable names or scopes. The inlining rules take care of translating local names to global names before typechecking occurs.

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

"$\mathbb{E}$ and $\mathbb{H}$ are solutions to $\mathtt{C}$"

As before we assume that the solver natively supports records and uninterpreted functions, but do not assume that the solver understands methods. Methods are inlined using the rules further below.

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

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

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

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

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

The rules for stay constraints are unchanged from Babelsberg/UID.

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

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

We use an inlining judgment to translate expressions into a representation suitable for the solver. 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.

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

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

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

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

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

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

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

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

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

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

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

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.

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

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.

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

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

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

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

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

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

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

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

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. We handle this in two phases: in a 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.

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

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

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

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

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

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|true>} \\ \texttt{<}\mathbb{E}'\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{s}_1\texttt{>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathtt{E}'\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|if e then }\mathtt{s}_1\texttt{ else }\mathtt{s}_2\texttt{>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathtt{E}'\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{>}} \text{(S-IfThen)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}\mathtt{e}\texttt{>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|false>} \\ \texttt{<}\mathbb{E}'\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{|}\mathtt{s}_2\texttt{>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathtt{E}'\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{>}\end{array}}{\texttt{<}\mathbb{E}\texttt{|}\mathtt{E}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|if e then }\mathtt{s}_1\texttt{ else }\mathtt{s}_2\texttt{>} \rightarrow \texttt{<}\mathbb{E}''\texttt{|}\mathtt{E}'\texttt{|}\mathbb{H}''\texttt{|}\mathbb{C}''\texttt{|}\mathbb{I}''\texttt{>}} \text{(S-IfElse)}$$

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

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

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

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

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathtt{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}\rangle \Downarrow \langle \mathbb{E}'|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathtt{v}\rangle \\ \langle \mathbb{E}',\mathtt{S},\mathbb{H}',\mathbb{C}',\mathbb{I}',\mathtt{e}_l.\mathtt{l}\rangle \rightsquigarrow \langle \mathbb{E}'',\mathtt{e}_c,\mathtt{e}'\rangle \\ \langle \mathbb{E}''|\mathbb{H}'|\mathbb{C}'|\mathbb{I}'|\mathtt{e}'{=}{=}\mathtt{v}|\mathtt{e}_c\rangle \Rightarrow \langle \mathbb{E}'''|\mathbb{H}''\rangle\end{array}}{\langle \mathbb{E}|\mathtt{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}_l.\mathtt{l}\ :=\ \mathtt{e}\rangle \rightarrow \langle \mathbb{E}'''|\mathtt{S}|\mathbb{H}''|\mathbb{C}'|\mathbb{I}'\rangle} \text{ (S-AsgnLValue)}$$

We do not allow updating the constraint store or heap during inlining. Thus, the above rule can only be used in constraint-construction mode if $\mathbb{H}{=}\mathbb{H}'$. The same is true for the rules below.

$$\frac{\begin{array}{c}\mathtt{C}_0 = \rho\ \mathtt{e} \qquad \langle \mathbb{E},\mathtt{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}\rangle \rightsquigarrow \langle \mathbb{E}',\mathtt{e}_{C_e},\mathtt{e}'\rangle \qquad \mathtt{C}_0' = \rho\ (\mathtt{e}' \wedge \mathtt{e}_{C_e}) \\ \mathbb{E}';\mathbb{H} \vdash \mathtt{C}_0' \qquad \text{i-stay}(\mathbb{E}') = \mathtt{C}_{S_s} \qquad \text{i-stay}(\mathbb{H}) = \mathtt{C}_{H_s} \\ \langle \mathbb{E}',\mathbb{H},\mathbb{I},\mathbb{C}\rangle \rightsquigarrow \langle \mathbb{E}'',\mathtt{C}\rangle \qquad \mathbb{E}''';\mathbb{H}' \models (\mathtt{C} \wedge \mathtt{C}_{S_s} \wedge \mathtt{C}_{H_s} \wedge \mathtt{C}_0')\end{array}}{\langle \mathbb{E}|\mathtt{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{once}\ \mathtt{C}_0\rangle \rightarrow \langle \mathbb{E}'''|\mathtt{S}|\mathbb{H}'|\mathbb{C}|\mathbb{I}\rangle} \text{ (S-Once)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathtt{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{once}\ \mathtt{C}_0\rangle \rightarrow \langle \mathbb{E}'|\mathtt{S}|\mathbb{H}'|\mathbb{C}|\mathbb{I}\rangle \\ \mathbb{C}' = \mathbb{C} \bigcup \{(\mathtt{S},\mathtt{C}_0)\}\end{array}}{\langle \mathbb{E}|\mathtt{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{always}\ \mathtt{C}_0\rangle \rightarrow \langle \mathbb{E}'|\mathtt{S}|\mathbb{H}'|\mathbb{C}'|\mathbb{I}\rangle} \text{ (S-Always)}$$

$$\frac{\begin{array}{cc}\langle \mathbb{E}|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}_0\rangle \Downarrow \langle \mathbb{E}_0|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{v}\rangle & \langle \mathbb{E}_0|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{e}_1\rangle \Downarrow \langle \mathbb{E}_1|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{v}\rangle \\ \langle \mathbb{E}_1,\mathtt{E},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}_0\rangle \rightsquigarrow \langle \mathbb{E}_2,\mathtt{e}_{C_0},\mathtt{e}_0'\rangle & \langle \mathbb{E}_2,\mathtt{E},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}_1\rangle \rightsquigarrow \langle \mathbb{E}_3,\mathtt{e}_{C_1},\mathtt{e}_1'\rangle\end{array}}{1\ \langle \mathbb{E}|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{once}\ \mathtt{e}_0\ ==\ \mathtt{e}_1\rangle \rightarrow \langle \mathbb{E}_3|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}\rangle} \text{ (S-OnceIdentity)}$$

$$\frac{\begin{array}{c}\langle \mathbb{E}|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{once}\ \mathtt{e}_0\ ==\ \mathtt{e}_1\rangle \rightarrow \langle \mathbb{E}'|\mathtt{E}|\mathbb{H}'|\mathbb{C}|\mathbb{I}\rangle \\ \mathbb{I}' = \mathbb{I} \bigcup \{(\mathtt{E},\mathtt{e}_0 = \mathtt{e}_1)\}\end{array}}{\langle \mathbb{E}|\mathtt{E}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\mathtt{always}\ \mathtt{e}_0\ ==\ \mathtt{e}_1\rangle \rightarrow \langle \mathbb{E}'|\mathtt{E}|\mathbb{H}'|\mathbb{C}|\mathbb{I}'\rangle} \text{ (S-AlwaysIdentity)}$$

4.3. Collections of Objects

Collections of objects semantics

4.4. Executable Specifications

The design of a formal semantics for a programming language ideally involves, besides creating the rules and proving that they satisfy the desired properties, executing the semantics in some form to run example programs and to study the details of the execution trees. A number of systems support practitioners in this, on of which is RML[**pettersson1994rml**]. RML is a programming language to generate executables from big-step semantics. It provides *relations* as basic building blocks for the semantics and translates these to C. We have implemented the semantics of Babelsberg/Objects in RML[**felgentreff+:2015:babelsbergrml**]; we created a parser for Babelsberg/Objects, implemented all the judgments and their

rules as RML *relations* and corresponding *rules*, and finally created an interface to the Z3 optimizing solver[**bjornernuz**] to implement the $\mathbb{E};\mathbb{H} \models \mathtt{C}$ judgment.

But even when the semantics is in executable form, it is not trivial to check implementations for conformance to the implemented semantics. This is regardless of whether the implementation evolved before or alongside the formal semantics or whether the implementation was created following the principles of the formalization. In either case, language specifics and optimization add complexity and make the conformance to the semantics harder to judge. Many languages, such as Ruby[1], Perl[2], or Java[3] provide an extensive test suite to check concrete implementations for conformance to a (sometimes informal) specification. These test suites include programs written in the implemented language and check their results. Babelsberg, however, is not a design for a single language, but for a family of languages—of which our semantics language Babelsberg/Objects may be seen as one. We thus provide a framework to transform the test suite for Babelsberg/Objects into test suites for the other implementation languages.

The tests for Babelsberg/Objects evolve alongside the semantics to clarify corner cases and document design decisions. As the semantics of Babelsberg evolves, we adapt and extend the suite. Given our executable semantics, we can immediately check if a change works as intended.

To also check if the actual implementations correspond to our semantics, we have created a framework to generate language-specific test suites. A new language implementer only needs to provide two modules: one for scaffolding that sets up the language specific testing environment and includes a placeholder macro which can be replaced by the framework with the concrete test case code, and a second module that implements source-to-source transformation. The scaffolding modules requires an implementation of the methods used in the semantics test cases available for calling from the tests, in particular for testing field equality, a mechanism to create records with variable numbers of fields. The source transformation module is highly language dependent, but straightforward. Each evaluation rule in the basic semantics—for the entire program, for statements, expressions, objects, and the operations on them—needs a rule that translates this operation into the target syntax.

- we provide a set of test cases that should work on all conformant implementations

- implementers can provide a src-to-src translation to generate specific test cases

- an executable semantics in RML implements our language and allows us to run these tests

[1] http://rspec.info/, retrieved Feb 25, 2015

[2] http://perl6.org/specification/, retrieved Feb 25, 2015

[3] http://openjdk.java.net/groups/conformance/, retrieved Feb 25, 2015

5. An Architecture For Using Multiple Constraint Solvers

In OCP, however, all solvers use the same interface to communicate with the VM so developers can add new solvers and replace existing ones.

Furthermore, the solvers embedded in CIP languages such as Kaleidoscope ignore encapsulation and manipulate the contents of variable storage directly. To include complex objects such as files in constraints, it may not be desirable to give the solver direct access to the file system.

In OCP, we require solvers call a hook method to set a new value on an object and only manipulate the object state directly if the object does not respond to the hook selector.

This precludes using constraints on objects that are not simply structures of primitive values, such as file or method objects.

- define the common interface we use for solvers

- explain in general what each solver needs to do, and how we manage variables that occur in multiple solvers

- explain how solver selection may work automatically

- based on types

- based on encountered operations+operands

- based on use of soft constraints

- solvers define capabilities, runtime discovers needs in constraints

- then show that now we have multiple solvers, the general interface needs to communicate

- explain why we did not choose SMT

- refer to the architecture for cooperating solvers

- we have made a change–variables are globally read-only, not just in one constraint, regions now allow using the same solver multiple times

Part III.

Implementations of Object-Constraint Programming

Given the specification and test suite for the Babelsberg design, we consider different implementation strategies for concrete Object-Constraint Programming languages. We implemented the Babelsberg design as a prototype in the Topaz[1] VM for Ruby[**flanagan2008ruby**], called Babelsberg/R. This implementation depends on a custom VM, but provides full integration with all types of variables in Ruby, the JIT, and the meta-object protocol. A second implementation in JavaScript, called Babelsberg/JS, is implemented as a library. This implementation runs on a variety of JavaScript VMs, at the cost of requiring a source transformation and not fully supporting local variables in constraints. A third implementation in Squeak/Smalltalk[], called Babelsberg/S, provides OCP as a library without requiring source transformation and including debugging support.

[1] TODO

6. Object-Constraint Programming as Language Extension

6.1. Goals

Performance, Full Integration

6.2. Implementation

The Ruby VM we use as a basis for Babelsberg/R is *Topaz* [**FelgentreffTopazWroclove2013**], an experimental VM built using the *PyPy/RPython* toolchain [**rigo2006pypy**]. While Topaz is a Ruby interpreter, the RPython toolchain provides it with a fast JIT compiler and garbage collector. We have extended the interpreter, and inherit the JIT and garbage collector from RPython. In this section, we first provide an overview of the key features of the implementation before plunging into the details.

The Topaz Ruby VM is written in a restricted, object-oriented subset of Python called RPython. Topaz executes Ruby by compiling Ruby source code into bytecodes and interpreting the bytecodes. The interpreter is written in an object oriented fashion, with each bytecode implemented as a method of the class `Interpreter`. Topaz also represents Ruby language constructs such as references and scopes as instances of RPython classes. As an example, local variables are represented by `Cell` objects, which provide methods to access and update the variable's value. In Babelsberg/R, we leverage the object-oriented design of Topaz to implement OCP.

The changes we made to the Topaz VM are two-fold. First, we modified the interpreter to support a *constraint construction mode* (cf. Section 6.2.2) and a primitive to enter this mode. Second, we extended the cells to support *constrained variables* (cf. Section 6.2.1) by allowing the same name to refer to multiple objects, one an object-oriented value and the other a *solver object*. (Note that this use of the same name to refer to multiple objects is a feature of the implementation only — it is not visible to the programmer.)

6.2.1. Constrained Variables

Ruby provides five types of variables: locals, instance variables, class variables, globals, and constants. (While constants should be assigned only once, this is

simply a convention in Ruby, and is not checked by the interpreter.) Of these variable types, we allow three as constrained variables: locals, instance variables, and class variables.

Ordinary variables are converted automatically to constrained variables if they are used in a constraint expression. A constraint expression is passed to the `always` primitive as an ordinary block closure. All locals referenced in the expression are stored as cell objects, which in Babelsberg/R have an additional field to store solver objects. Instance and class variables in Topaz are stored in maps [**chambers1989efficient**]. For Babelsberg/R, we have extended these to store solver objects as well as ordinary objects, so that once the JIT has warmed up, read access to solver objects is no slower than access to ordinary objects.

Converting ordinary variables into constrained variables has an impact on garbage collection. Solver objects usually have more references pointing to them than ordinary objects — besides their scope and owner (in the case of instance or class variables), solver objects are also referenced from one or more constraints. Because solver objects can also be implemented in the host language, this means that they may only be garbage collected if no more active constraints refer to them.

6.2.2. Execution Contexts

To implement the different execution modes — *imperative execution*, and *constraint construction* — we extended the default interpreter, and added `ConstraintInterpreter` as a subclass. The `ConstraintInterpreter` changes how locals, instance variables, and class variables are accessed.

6.2.2.1. Imperative Execution

This is the normal execution mode, as implemented in the Topaz interpreter. We have extended the STORE and LOAD bytecode methods with an additional check whether a variable refers just to an ordinary object or to a solver object as well. In the latter case, instead of reading or changing the value of the variable, the solver object for the variable receives the messages `value` or `suggest_value`, respectively. As long as all constraints are satisfied, this difference is not visible to the programmer. However, while direct, destructive assignment to a normal variable in imperative programs always succeeds, `suggest_value` triggers one or more solvers. If any solver fails to satisfy its constraints using the new value, an exception is raised. This exception is propagated by the VM, and should either be handled by the programmer or allowed to halt execution of the program. If an assignment fails in this way, the variable retains its original value.

As described in **??**, in Babelsberg/R, if multiple variables that have constraints on them are assigned using multi-assignment, all values are assigned before a solver is triggered, and all assignments are undone if an exception is raised during solving. This way, if the exception is caught, the program is not in violation of any constraints.

6.2.2.2. Constraint Construction

The initializer for constraint objects — usually called through `always` — activates an execution context for constraint construction. In this mode, variable values are replaced with their associated solver objects in LOAD instructions. These solver objects are created by sending `for_constraint` to the current variable value. This method should return an object that responds to `value` and `suggest_value`. The `value` method extracts the value from a solver's internal representation, while `suggest_value` sends the candidate value to a solver and requests that it re-satisfy the constraints. For values that do not respond to `for_constraint`, a generic solver object is returned that removes and recalculates all constraints in which it occurs whenever its value changes.

STORE instructions in this mode create equality constraints. This is necessary to support constructing new objects in the predicates that connect values (for example, a 2d point with x and y values that were constructed in the constraint.) However, this also means that all code blocks encountered during constraint construction must be in single assignment form (cf. **??**).

To support solvers written in the host language, the VM needs a way to distinguish code that should be executed in this mode from code that should not. To support this, solvers written in the host language should be subclasses of `ConstraintObject`. This class serves as a marker to leave constraint execution mode when we enter solver code.

Constraint Solving This mode is simply a flag that is active whenever code runs in the dynamic extent of a method of a `ConstraintObject`. It prevents nested sends of `suggest_value` from causing recursive calls to the solver. This is necessary to support solvers written in Ruby, but implies that solvers themselves cannot use constraints in their implementation.[1]

Constraint Construction Example

To illustrate how our implementation supports the combination of objects and constraints, consider the rectangle example in Babelsberg from **??**. The code asserts that the area of the rectangle `rect` should always be greater than or equal to 100. The assertion is expressed by sending the `area` method to `rect`, and then sending the `>=` method to the result. The only variable named explicitly in the constraint is `rect`, but there are other variables that play a role in it.

In constraint construction mode, the `rect` variable is replaced with a generic solver object, since the class `Rectangle` does not implement the method `for_constraint`. This placeholder delegates the message `area` to the rectangle. During the execution of the `area` method, the points are replaced with generic solver objects, and their x and y values (floats) are replaced with specific solver objects (for example Z3 variables, cf. **??**). In this mode, the messages to the float values return symbolic expressions rather than calculating the current area of the rectangle. The expression

[1] If we wanted to support other cooperating paradigms in addition to constraint-oriented programming, we would generalize this technique to support additional execution contexts.

representing the area of the rectangle is then sent the message >= with 100 as its argument and returns an inequality constraint.

The constraint construction is complete when the block passed to `always` returns. The values and relations among them produced by this symbolic execution are gathered into a *Constraint* object containing specific solver objects and generic solver objects. Specific solver objects are objects that are connected in a way that the solver can reason about (e.g., floats connected via equalities or inequalities or arithmetic relations, or booleans connected via boolean operations). Generic solver objects are all other objects that contribute to the constraint but about which the solver cannot reason directly.

The constraint solvers operate on specific solver objects directly to solve the constraint. Changes made by the solver to these objects show up in the object-oriented (OO) view on the next access to the variables from imperative code, at which point their values are copied to their OO counterparts. Assignments to any variable encountered during the constraint construction will trigger the solver to re-satisfy the constraint (and potentially raise an error if the assignment is inconsistent with the constraint).

Generic solver objects invalidate the constraint if their OO value changes through imperative assignment. This invalidation retracts all solver objects created during constraint construction and re-executes the block to create new solver objects and constraints. This means that the block may be re-executed multiple times during the run-time of the program. (This is one reason why any side-effects in constraints should be benign, as noted in **??**.)

6.2.3. Implementing Edit Constraints

Edit constraints are used to support incremental constraint satisfaction, and are important for achieving good performance in interactive applications. The `edit` method provided as part of Babelsberg/R's Cassowary library adds edit constraints and repeatedly updates them with values from a stream.

Cassowary as shipped in the Babelsberg/R standard library allows the variables and stream to hold user-defined objects as well as primitive types. The library uses the application programming interface (API) of the `Constraint` class to access the constraint values associated with variable names, creates edit constraints for them, and feeds the stream into the resulting edit variables.

To use Cassowary as the solver, we need edit variables that are Floats (e.g., the `x` and `y` values of a midpoint), but we also want to do this in an object-oriented way that respects encapsulation. To support this, the client passes an array of method names for the return values that should be updated in the edit constraint (e.g., `x` and `y` for a point — those values may be calculated or direct accessors.) Cassowary creates new edit variables, and adds an equality constraint to the return values of the methods. Thus, the internal storage layout of the class is not visible to the programmer from outside the object, because the equality constraint is simply asserted on the results of message sends using the `always` primitive.

In the following example, the mouse locations or the mouse point might store their x and y values directly, or might be points represented using polar coordinates. In either case, the edit constraints apply to the return values of their respective x and y methods:

```
edit(stream: mouse.locations.each, accessors: [:x, :y]) { mouse_point }
```

In a DeltaBlue-specific edit method, the edit constraints returned could be simpler, since DeltaBlue local propagation methods can apply to user-defined objects such as points, not just to floats. The point would be simply updated rather than dealing with its x and y coordinates separately, and the data flow plan would update the objects constrained to be equal to the point that represents the mouse location.

6.2.4. Adding New Solvers

During constraint construction, the VM sends the `for_constraint` message to each variable value encountered during the execution. User code can add solvers to the system by dynamically adding a `for_constraint` method to those classes for which the solver is applicable, making use of Ruby's open classes. This method takes the name under which the variable is accessed as an argument, and should return an object that implements a subset of the interfaces that the solver can reason about, as well as the methods `value` and `suggest_value` (as described in cf. Section 6.2.2.)

The Cassowary solver extends the Float class:

```
def for_constraint(name)
  v = Cassowary::Variable.new(name: name, value: self)
  Cassowary::SimplexSolver.instance.add_stay(v)
  v
end
```

This method creates a new variable, adds a low-priority stay so the solver attempts to keep the value stable, and returns the constraint object. The VM then sends messages to this object instead of the Float object in the context of the constraint execution.

In the case of immutable objects (such as floats and integers), the VM directly returns the variable value determined by the solver. However, for mutable objects, solver libraries can provide the method `assign_constraint_value` to update the ordinary object. Babelsberg/R provides a solver over numeric arrays, which uses `assign_constraint_value` to update the array contents. In the `for_constraint` method for arrays, it returns a `NumericArrayConstraintVariable` if all elements in the array are instances of a subclass of `Numeric`.

```
class Array
  def for_constraint(name)
    if self.all? { |e| e.is_a? Numeric }
      return NumericArrayConstraintVariable.new(self)
    end
  end
```

```
  def assign_constraint_value(val)
    self.replace(val)
  end
end
```

This illustrates how the solvers provided by Babelsberg/R are simply constraint solver libraries that extend core classes. The `NumericArrayConstraintVariable`, for example, allows element access, provides the Ruby collection API (each, map, inject, ...), the `sum` method (which calculates the sum of elements) and the message `length`, to solve constraints over the length of arrays.

Below is an example that asserts constraints on TWP-encoded[2] short strings (represented as an array of bytes). TWP short strings are at most 110 bytes, with the first byte giving the length of the string plus a tag. Line 3 constrains this particular string to contain only capital letters (the `?A` Ruby syntax gives the byte value of a character).

```
twp_str = []
always { twp_str.length == twp_str[0] + 17 }
always { twp_str.length <= 109 }
always { twp_str.each {|byte| byte >= ?A \&\& byte <= ?Z} }
```

[2] http://www.dcl.hpi.uni-potsdam.de/teaching/mds/twp3.txt

7. Object-Constraint Programming as a Library

7.1. Goals

- Avoid limitations of in-VM approach
  - Allow richer tools
  - easier introspection support
  - Available everywhere
  - Toolbox approach
  - Don't pay for what you don't use

7.2. Implementation

7.2.1. Babelsberg/JS

We have implemented Babelsberg/JS in the Lively Kernel environment [**lively-kernel-2008**]. We provide pure JavaScript implementations of DeltaBlue and Cassowary as constraint solvers and extend the Lively Kernel JavaScript interpreter to evaluate constraint expressions. The code is not Lively specific – we use the collection APIs and class system of Lively, but this could be trivially changed. However, when used in the Lively environment, we provide a source transformation that makes writing constraints in the Object Explorer [**lively-partsbin-2012**] more convenient.

7.2.2. Assignment

Assignment to objects that are constrained in Babelsberg/JS is the core concept that binds the declarative constraints and imperative code together. Whereas in standard imperative code an assignment writes a value to a memory location, assignments in Babelsberg add equality constraints on constrained objects and trigger re-satisfaction. The new equality constraint may be unsatisfiable, in which case the assignment is not executed and a runtime exception is generated.

As in Babelsberg/R, the Babelsberg/JS runtime informs the developer of a failed assignment by generating a runtime exception. To support the cooperating solvers design, assignment in Babelsberg/JS is a 3-step process:

Set Value If the new value is the same as the old, we simply return. Otherwise, we convert all read-only constraints on the assigned variable either to required edit constraints (for solvers that support them) or to equality constraints.

If the assigned value has an external variable, i.e., it is constrained by a solver that can handle its type (for example a real in Cassowary), the new value is input into the furthest upstream solver using an equality constraint and this solver is then called. Afterwards, the equality constraint for assignment is removed. However, if the solver cannot satisfy its constraints with the new value, an exception is raised.

Update Downstream Variables For all external variables except the primary one (the one in the most upstream solver), the new value is input into the solver. If any of the solvers fail to satisfy their constraints with the new value, an exception is generated and all read-only constraints are re-enabled as above.

All remaining constraints are in solvers that cannot handle the type of that variable. Consequently, its value was treated as a constant in their constraint expressions. With the new value, these have to be recalculated. The old variable value is remembered and the new value is stored. These constraints are disabled, their expressions re-evaluated in constraint construction mode, and then the constraints are re-enabled. If any constraint in this set fails to run its expression or cannot be satisfied with the new value, the old value is restored and an exception is generated.

Update Connected Variables Finally, we have to update the variables connected to the assigned variable. To do so, we create the transitive closure of all variables connected to the assignee through constraints. For all these variables, a new value has already been created for all solvers that already ran, but their downstream read-only constraints still have to be updated. These variables have to go through the first two steps of the assignment process, returning early if their values have not changed.

At this point, assignment can only fail for variables that are in solvers that have not run yet. These are only solvers that the primary assignee is not part of. If any one of these solvers cannot satisfy their constraints with the new value, we restore the old value, re-satisfy the constraints, and raise an exception. While this may leave the system in a different state than it was in before assignment (depending on the implementation of the participating solvers, they may not deterministically find the same solution to the same set of constraints) the system will still be in a state that satisfies all previous constraints.

Deferred Assignment of Connected Variables Babelsberg/R included an optimization to defer copying the values from a solver to the object-oriented variable location after assignment. Instead of copying the values for all affected variables immediately, the variable's values would be copied when they are next used in imperative code. This optimization cannot be used with our cooperating constraint solver architecture. Consider the following contrived example:

```
var obj = {a: 10, b: 10, c: true};
always: { solver: cassowary
   obj.a + obj.b == 20
}
always: { solver: deltablue
  obj.c.formula([obj.b], function (b) {
    if (b == 13) throw "unlucky";
    return b < 10;
  })
}
obj.a = 7;
```

When `obj.a` changes, Cassowary is called to resatisfy the first constraint. However, to trigger DeltaBlue to solve the second constraint, the new value for `obj.b` has to be copied immediately, rather than when `obj.b` is next read. Otherwise, a failure to satisfy the second constraint is only encountered sometime later in the execution and difficult to trace back to the assignment that caused it.

Assigning Mutable Objects So far we have described how assignment is handled for atomic objects, such as integers and floats. We have not, in our design, addressed the case of mutable objects with substructure. Consider the following midpoint line:

```
var line = {start: pt(0,0), end: pt(1,1), midpoint = pt(0,0)};
always: { solver: cassowary
  var center = line.start.getPosition().
              addPt(line.end.getPosition()).scaleBy(0.5);
  line.midpoint.getPosition().eqPt(center);
}

line.midpoint = pt(1, 1);
```

There are two ways to look at such an assignment: a) the assignment asserts equality between `midpoint` and `pt(1, 1)` — both mutable objects — not their `x` and `y` parts. So the solver could also modify the parts of the newly assigned point to satisfy the constraint. This seems counter-intuitive, so presumably the right-hand side of an assignment should be read-only to the solver. On the other hand, there are use-cases for constraints for example in input rectification [**long-rectification-2012**], where programmers may expect the system to fix the assigned object, rather than reject the assignment.

For now, we consider the behavior in this case to be implementation defined. Babelsberg/JS marks the assigned objects' parts with strong stay constraints. This means that, as long as other constraints allow, the solver will not change the new position for the midpoint. The design of a general solution is subject of further research.

7.2.2.1. Changing the Type of Variables

Most solvers only provide support for a limited number of type domains (such as reals or booleans). When variables are used in constraints, their current values

determine how they are handled by the solvers. Changing the type of a variable, although possible in a dynamic language, is a relatively uncommon operation, so slow performance is acceptable. When it does occur, the variable is removed from all solvers, all its constraints are disabled, and its constraint expressions are re-executed in constraint construction mode, thus creating new solver-specific representations.

7.2.3. Constraint Construction

When a programmer writes a function that contains a constraint expression, this expression is evaluated using our JavaScript *ConstraintInterpreter*. Popular JavaScript VMs[1] (Apple Safari's SquirrelFish[2], Google Chrome's V8[3], Mozilla Firefox's SpiderMonkey [**gal-tracejit-2009**], or Microsoft's Chakra[4]) do not provide direct access to the native interpreter or execution context of the caller, so our interpreter cannot look up names used in the constraint expression in the caller's environment. Instead, those names have to be passed explicitly.

Babelsberg/JS provides a source-to-source transformation based on *UglifyJS*[5], which collects names from the context and modifies the source code to pass those names into the constraint expression. This source transformation is enabled automatically when programmers use Babelsberg/JS in the Lively Kernel's Object Editor. For other JavaScript code, they have to provide a context object explicitly.[6]

Given a context, a function, and a solver, the ConstraintInterpreter executes the expressions in the function to create the constraint. The ConstraintInterpreter subclasses a JavaScript interpreter, modifying its behavior in three main aspects:

1. Slot accesses are intercepted. For each slot accessed during the execution of a constraint expression, property accessors are created that delegate access to a *ConstrainedVariable* object. For each slot, only one ConstrainedVariable is created on first access. ConstrainedVariables manage the communication with the various solvers and create solver specific representations of the slot value.
2. Certain unary (`!` and `-`) and binary operations (arithmetic, equality, inequalities, conjunction) are not interpreted as usual if an operand is a ConstrainedVariable or an expression involving ConstrainedVariables. Instead, the constraint object is sent a message to construct a solver-specific expression representing the operation and that expression is returned. For example, in Cassowary, the expression `a.value <= b.value` would return a *LinearInequality* object.

[1] http://www.w3schools.com/browsers/browsers_stats.asp
[2] http://trac.webkit.org/wiki/SquirrelFish
[3] https://code.google.com/p/v8/
[4] http://en.wikipedia.org/wiki/Chakra (JScript engine)
[5] http://lisperator.net/uglifyjs/
[6] Note that we cannot use `eval` to access the outer scope. If we only supported constraints that access fields of objects in the scope and do not call user defined functions, we could have rewritten the code and evaluated the constraint expression using JavaScript's `eval` function, which has access to the enclosing scope. Using a custom interpreter, however, allows us to easily instrument the execution of most user-defined functions, so we can use normal object-oriented methods in constraint expressions.

3. Functions invoked in the expression are also interpreted in the Constraint-Interpreter by default. However, the plain JavaScript interpreter is used if the receiver is a ConstrainedVariable. In that case, the call is executed using normal JavaScript execution semantics. This is required to avoid creating constraints on the state of the solvers themselves.

The responsibility of ConstrainedVariables during constraint construction is to pass calls to the appropriate solver. To that end, a ConstrainedVariable lazily builds a mapping from solvers to solver-specific representations of its value. During construction, if the programmer has explicitly selected a solver, this solver is asked to provide a value representation by sending the message `constraintVariableFor` with the value as argument. If no solver was provided, the value is sent the `constraintSolver` message. Solver libraries may override this message for types that they can operate on. If the value responds with a solver instance, this solver becomes the active solver for the currently constructed constraint and is asked to provide a representation, again by sending `constraintVariableFor`.

Whenever a new representation is created in this manner, the solvers are sorted to determine which region the variable belongs to. Only the solver responsible for this region may write to the variable; as far as all other solvers are concerned it is read-only.

7.2.4. Determining Cooperating Solver Regions

The architecture for cooperating constraint solvers requires that each variable must be read-only in all but one of the regions that it occurs in. Furthermore, the regions and associated solvers must form an acyclic graph.

In Babelsberg/JS, when a variable appears in a new solver, we gather the solvers for the variable and sort them into regions. The region information is stored as a property of a solver instance. This allows, for example, the use of multiple instances of the same solver in different regions.

The variable is marked read-only for all solvers except the one in the furthest upstream region, the *defining solver*. This means that new values are assigned by calling `suggestValue` on the defining solver, and that all other solvers are triggered (in descending order of regions) once the defining solver has resatisfied its constraints, as described in Section 7.2.2.

7.2.5. Edit Constraints

Since the original Babelsberg design did not include language-level support for edit constraints, these were supplied by the solver libraries. In Babelsberg/R, the meta-level protocol for inspecting constraints was used to support edit constraints in Cassowary and DeltaBlue. The programmer called the appropriate edit method with the objects to be edited and a stream that would provide new values. The Constraint meta-protocol was used to create edit variables, constrain them to be equal to the supplied variables, and update them from the stream.

To support cooperating incremental re-solving (cf. **??**), in Babelsberg/JS there are two changes to this scheme. First, to support edit constraints within a single thread, the edit method returns a callback to input new values into the solvers, rather than taking a stream of values. Second, since the language design now supports edit constraints explicitly, the solvers have to provide a specific edit constraint API.

Upon calling the edit method, the following methods are called on the solvers and the supplied variables, in order:

prepareEdit is called on each solver variable. In this method, variables can prepare themselves for editing. In Cassowary, for example, this would call the `addEditVar` method on the solver with the variable as argument. For DeltaBlue, this creates an `EditConstraint` on the variable and adds it to the list of constraints.

beginEdit is called once for each solver participating in the edit before the callback is returned. In Cassowary, this initializes the edit constants array and prepares the solver for fast re-solving when these constants change. In DeltaBlue, the solver generates an execution plan to solve the constraints starting with the EditConstraints as input.

Now the callback can be used to input new values into the system and trigger re-solving. The callback will call **`resolveArray`** on each solver with the new values and update the object's storage (so other observers and hooks around the values still work). Because the solver's execution plan is fixed for the duration of an edit, we disallow creating new edit callbacks before the current edit has finished. When new constraints are created, the execution plan may also become invalid, but we do not enforce invalidating the edit callback in this case.

To finish editing, the callback is simply called without supplying new values.

finishEdit is sent to each solver variable. Cassowary variables do nothing here, DeltaBlue variables remove their EditConstraints from the solver.

endEdit is called once for each solver to reset the solver state.

Compared to Babelsberg/R, this makes the interface for edit constraints uniform across solvers and also allows it to work with cooperating solvers. However, each solver now has to provide some support for this, so more work is required to enable the feature.

7.2.6. Babelsberg/S

8. Discussion of Implementation Approaches

8.1. Complexity of the Design

- Host Language requirements (intercession)
  - in-language interpreter+parser
  - custom vm
  - limitations on syntax (?)

8.2. Performance Analysis

- comparison against in-language solutions
  - benchmarks when using/not-using

Part IV.

Applications with Object-Constraint Programming

9. Writing Constraint Code

Using our prototype implementations of this design, we have written a number of applications, including some from the domains of load balancing, electrical and mechanical simulation, puzzle solving, temperature conversion, and layouting.

9.1. Idioms

in-initializer-constraints, constraint-manager-class, local variables/readonly

9.2. Control Structures

9.2.1. Scoped Constraints

9.2.2. Constraint Layers

9.2.3. Constraint Triggers

10. Understanding Constraint Code

10.1. Debugging

Debugger

10.2. Introspection

Inspector, Highlighter, Constraint-Graph

Part V.

Discussion and Conclusion

11. Related Work

Structured as a sort of "History of constraint programming"

11.1. Constraint Programming Systems

11.2. Constraint Logic Programming

11.3. Constraint Imperative Programming

11.4. Constraint Programming Libraries and DSLs

11.5. Constraints in General Purpose Programming

12. Summary and Outlook

12.1. Discussion

12.2. Future Work

We consider the work presented here to be a useful tool in a general purpose programming language. Many avenues for future research present itself, both small and large. Among the smaller issues left for future work are questions around debugging, comprehension, and the choice of solvers.

For debugging and visualization of the constraint interpretation and solving process, we have presented a debugger that shows information about how constraints are constructed and satisfied. This integration treats the solver as a black box, but shows which objects are affected by constraints, which constraints are relevant in a given scope, and how value changes are triggered by constraint satisfaction. While this can help in understanding *when* and *how* the solver affects the system, it does not explain *why*. In future work we will explore options to visualize the solving process itself, step through the decisions of the solver, and allow the programmer to explore alternative solutions.

Our semantics provides clear rules as to what does and does not work in a constraint expression, however, additional guidance may be desirable during development to show where these rules are violated. This includes checking of constraint expressions and, if they are invalid, information about how it may need to change to work as part of a constraint. We currently provide error messages if some operations cannot be translated to the solver, and try to offer alternatives — for example, Cassowary does not support true inequality relations like $<$, but it may be possible to express the problem using $\leq$. Future work should could these messages to offer additional alternatives or the solver.

Our current implementations mostly leave the choice of solver up to the developer, and the semantics ignore the choice of solver completely. In cases where the developer does not know or care about which solver to choose, we use a simple scheme that tries all available solvers and chooses the first that returns a solution. More research is required to find better ways to automatically choose the right combination of solvers based on a variety of metrics such as solution quality or performance. The metrics itself are not clear as of this point, and might vary based on the needs of the developer, suggesting that an interactive dialog between the development environment and the developer might be required to choose the right solver.

We feel that a number of questions remain that are less clearly defined. In this work we position constraints as a tool in the utility belt of general purpose programming, and provide several control structures and patterns to scope constraints, their integration with existing module systems is not clear. Other questions concern constraints over time. It would be interesting to be able to express constraints over the relation of different subsequent system states. These are useful, for example, to express simulations in terms of differential equations, and have the solver ensure that the system changes at the right rate. Another idea is to express constraints over distributed systems, where synchronous solving is not feasible. Finally, constraints provide the potential to encode properties of the system that the developer implicitly assumes. It may be interesting to explore how an interactive environment can, through observation of direct manipulation or a form of interactive dialog with the developer, derive new or refine existing constraints, and thus help generalizing a concrete example into a general program.

12.3. Conclusions

Publications

Journals

- **FelgentreffJOT14**

Conferences

- **SteinertDLS2014**
- **FreudenbergDLS2014**
- **FelgentreffECOOP14**
- **PerscheidWCRE2014**
- **HirschfeldJSSST14**

Workshops

- **GraberREBLS2014**
- **TaeumelCOP2014**
- **FelgentreffWASDETT2013**

Technical Reports

- **felgentreff:2014:semantics**
- **felgentreff+:2014:babelsberg**

Part VI.

Appendix

13. First Unimportant stuff.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

14. Curriculum Vitæ

Todo list