The Design and Implementation of Object-Constraint Programming

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

21. April 2017

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 21. April 2017

Tim Felgentreff

Abstract

Constraints allow developers to specify properties of systems and have those properties be maintained automatically. This results in compact declarations to describe interactive applications avoiding scattered code to check and imperatively re-satisfy invariants in response to user input that perturbs the system. Constraints thus provide flexibility and expressiveness for solving complex problems and maintaining a desired system state. Despite these advantages, constraint programming is not yet widespread, with imperative programming still being the norm.

There is a long history of research on constraint programming as well as its integration with general purpose programming languages, especially from the imperative paradigm. However, this integration typically does not unify the constructs for encapsulation and abstraction from both paradigms and often leads to a parallel world of constraint code fragments and abstractions intermingled with the general purpose code. This impedes re-use of modules, as client code written in one paradigm can only use modules written to support that paradigm — thus, modules require redundant definitions if they are to be used in both paradigms. Furthermore, clear distinction between the paradigms requires developers to learn about and fully understand both paradigms to make use of them.

In our work, we have developed a design for a family of object-constraint languages called Babelsberg. Our design unifies the constructs for encapsulation and abstraction by using only object-oriented method definitions for both declarative and imperative code. Just like assertions, our constraints are expressed using ordinary imperative expressions, including full objects and message sends. Unlike assertions, however, the system attempts to satisfy them if they are not currently true, and keeps them satisfied throughout the remaining execution. We provide a semantics that guides implementers of our design to combine Babelsberg with existing object-oriented host languages both semantically and syntactically and to demonstrate its feasibility with an executable semantics and three concrete implementations of Babelsberg. To allow developers to use the power of constraints without having to understand the specifics of different constraint solving strategies, we integrate an architecture for using multiple cooperating solvers. Finally, based on our experience with the concrete implementations, we propose performant implementation strategies of key features for object-constraint programming.

We argue that our approach provides a useful step toward making constraint solving a useful tool for object-oriented programmers. We also provide example code, written in our implementations, which uses constraints in a variety of application domains, including interactive graphics, physical simulations, data streaming with both hard and soft constraints on performance, and interactive puzzles.

Zusammenfassung

Die Constraint-basierte Programmierung erlaubt es Entwicklern, erwünschte Eigenschaften eines Systems zu spezifizieren, und es dem System selbst zu überlassen, diese aufrechtzuerhalten. Dies resultiert in kompaktem, deklarativem Quelltext. Dies steht im Gegensatz zu verstreutem Quelltext, welcher bei imperativer Programmierung oft nötig ist, um Invarianten zu überprüfen und wiederherzustellen. Constraints bieten in diesem Sinne Flexibilität und Ausdruckskraft um komplexe Probleme zu lösen und erwünschte Systemzustände zu erhalten. Trotz seiner Vorteile ist die Constraint-basierte Programmierung jedoch nicht weit verbreitet, und die imperative Programmierung ist heute die Norm.

Constraint-basierte Programmiersprachen sowie deren Integration mit Allzweckprogrammiersprachen, insbesondere aus dem imperativen Paradigma, haben eine lange Forschungsgeschichte. Allerdings beschränkt sich die Integration bisheriger Arbeiten zumeist auf reine Interoperabilität der Paradigmen, statt deren Abstraktionsmechanismen zu vereinheitlichen. Das führt dazu, das Constraint-Quelltext und imperativer Quelltext vermischt werden, aber nicht einheitlich benutzt und entwickelt werden können. Das schränkt die Wiederverwendbarkeit von Modulen ein, da diese explizit für das eine oder andere Paradigma optimiert werden müssen, oder redundante Definitionen erfordern, um in beiden Paradigmen einsetzbar zu sein. Desweiteren zwingt der scharfe Bruch beim Paradigmenwechsel die Programmierer dazu, beide Paradigmen voll zu verstehen, bevor sie sinnvoll eingesetzt werden können.

In unserer Arbeit haben wir ein Design für eine Sprachfamilie von Objekt-Constraint-basierten Sprachen entworfen, welches wir *Babelsberg* nennen. In unserem Design sind die Konstrukte zur Abstraktion vereinheitlicht, indem nur objektorientierte Abstraktionen und Methodendefinitionen für sowohl imperativen als auch deklarativen Quelltext verwendet werden. Genau wie imperativer Fehlerprüfcode können Constraints nun objektorientierte Sprachkonstrukte verwenden, und das System kann diese in eine deklarative Form übersetzen und für den Rest der Laufzeit kontinuierlich prüfen und sicherstellen. Zu diesem Design präsentieren wir außerdem eine Semantik, welche Sprachentwicklern als detaillierte Grundlage zur Integration unseres Designs mit existierenden, objektorientierten Sprachen dienen kann. Die Korrektheit dieser Semantik haben wir für Schlüsseltheoreme theoretisch bewiesen, sowie praktisch mithilfe einer ausführbaren Semantik gezeigt. Mithilfe dreier konkreter Implementierungen unseres Designs zeigen wir weiterhin dessen Praktikabilität. Um Entwicklern aus dem imperativen Paradigma den Einsatz von Constraints zu ermöglichen, ohne dass diese die spezifischen Limitierungen der verschiedenen automatischen Constraint-Lösungsstrategien verstehen müssen, präsentieren wir außerdem eine Architektur die es ermöglicht, automatisch verschiedene Lösungsstrategien zu kombinieren, um auch komplexe Probleme zu lösen. Schlussendlich diskutieren wir anhand der konkreten Implementierungen deren Ausführungsgeschwindigkeit unter verschiedenen Bedingungen, sowie Erfahrungen in der Entwicklung und Benutzung von Objekt-Constraint-basierten Sprachen.

Wir glauben, dass unser Ansatz einen nützlichen Schritt darstellt, um das automatische Lösen von Constraints zu einem nützlichen Werkzeug für imperativen Programmierer zu machen. Das zeigen wir auch anhand von Beispielanwendungen, in denen wir unsere Implementierungen des Babelsberg Designs in einer Reihe von Anwendungsdomänen wie interaktive Grafiken, physika-

lische Simulationen, Datenübertragung mit Constraints über die Performance, sowie interaktive Puzzlespiele.

Acknowledgements

I want to thank Robert Hirschfeld and Michael Perscheid for convincing me to pursue a PhD. I would never have considered it before, but their support during my Master's studies made it an easy decision. I also want to thank Alan Borning who happened to visit at the right time for me to latch on to a topic that would have been much harder for me to get into without his help.

I want to thank my friends and current and former colleagues during my studies at HPI, in the Software Architecture Group, and the HPI Research School: Marcel Taeumel, Jens Lincke, Tobias Pape, Philipp Tessenow, Stephanie Platz, Konstantin Haase, Maria Kaline, Frank and Lysann Schlegel, Lars Wassermann, Anton Gulenko, Lauritz Thamsen, Bastian Steinert, Carl Friedrich Bolz-Tereick, Stefan Lehmann, Marco Roeder, Lena Feinbube: You all listened to my ideas and helped me put them into perspective at the right times, you pushed back when I went overboard and cheered me up when I was glum, you made my travels immensely enjoyable and memorable, you helped me without asking questions when I obviously procrastinated on random side projects, you introduced me to more board games than I can remember, and you tolerated me when I went ran into your offices singing at the top of my lungs. You shaped me as a person, and a part of each of you is forever a part of me.

The financial support, the freedom to explore, and the superb work environment that the HPI has afforded me through the HPI research school was fundamental to my work. I want to express my gratitude to Andreas Polze and Sabine Wagner for running the research school as you did while I was there.

I also want to thank my foreign friends who afforded me a wider perspective on the world during my PhD studies, who read and discussed my work with me, and who called ambulances for me and came with me to hospitals in the middle of the night: 蒋澜, 牛莅芃, Joshua Cape, Tanvi Shroff. I might not be here today without you.

I am grateful to my entire family for their continued support, for cooking and making coffee for me when I had to work through the night, for always believing in me and telling me so when I faltered, and for forgiving me when I was too preoccupied with other things that I wasn't there for them.

Finally, I could not have done this without my wife and my best friend Felicia Flemming, who was and is supporting me in everything I do, and who is helping me become as best a person as I can be.

Contents

List of Figures

List of Tables

Part I.

Solving Constraints on Object Behavior

1. Introduction

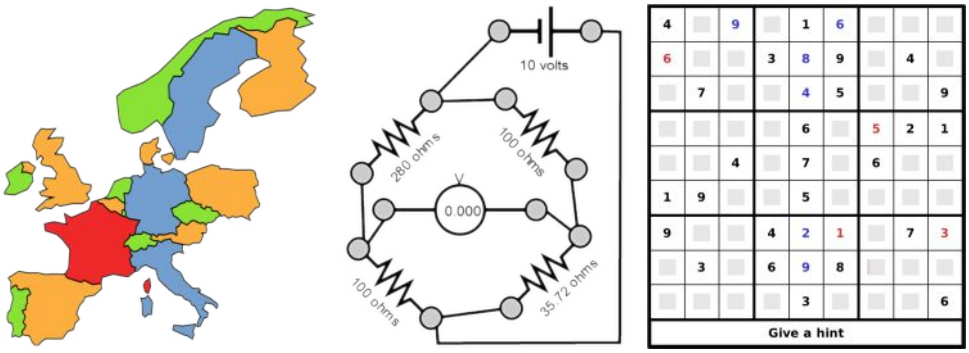
Imperative programming has become the dominant paradigm, because it works well in modeling properties of the real world such as destructive state changes or taking actions towards a specific goal. It is also traditionally close to the way physical computers work, and programmers have been (and still are) trained to approach new tasks or problems by thinking in terms of step-by-step instructions.

However, best-effort computing, the ability to parallelize algorithms, and cheap-but-unreliable hardware are becoming more important for a variety of application domains. We increasingly discover that under these circumstances, using only imperative code can make software harder to develop and understand, to extend and reuse, and to maintain and debug. While another paradigm offers clear advantages to approach a particular sub-problem in such domains, imperative programmers have to revert to using it through libraries or Domain-specific Languages (DSLs), or even (re-)write parts of their program in an entirely separate language . Since this requires additional investment in learning a new language or library, alternative workarounds and "best practices" proliferate to express problems imperatively that may be more easily expressed in another paradigm. Some such problems can be more elegantly expressed using *declarative, constraint-based programming*.

In contrast to imperative programming, constraint-based programming describes only a desired relation that should hold, not how to achieve this. Rather than obscuring the meaning of a piece of code with step-by-step instructions, a declarative description of the problem is handed to a constraint solver which then determines how to achieve (and possibly maintain) a desired outcome. Constraint programming thus provides flexibility in the solving approach, and avoids scattering code throughout the program to ensure that constraints are observed during execution.

Constraints offer some well known advantages over an imperative programming style. In a number of application domains relevant to computer science they often lead to more concise and comprehensible code. Such domains include graphical layouting, data structure repair, physical simulations, load balancing, or interactive puzzles. For these, it is often easier to state *what* the rules are rather than *how* to achieve them. Besides requiring fewer lines of code, the solving strategy may be exchanged without having to adapt the constraint declarations, and thus performance or quality of the solution can be tweaked easily. For problems that lend themselves well to using constraints, the code closely mirrors the desired solution, and enables developers to understand the goal of the program, even if the domain or the steps to achieve this goal were not previously known to them. Finally, constraint solvers have a clean semantics when it comes to combining multiple and possibly competing constraints, so the solutions to conflicts are clearly defined rather than hidden in the program control flow. Due to these advantages, many constraint-programming systems have been proposed over the years, with Sketchpad [115] in 1963 and ThingLab [8] in 1981 as early examples.

The limits of constraints, on the other hand, have often been understated, with constraint programming sometimes declared to be the "holy grail" of programming: programmers supposedly state their problem and the computer solves it [50]. In reality, while constraint solvers continue to advance, they are by no means able to solve general problems to produce any kind of desired outcome. In comparison to this lofty declaration, real constraint solvers are still relatively simple.



(a) Map layout and coloring (b) Circuit simulation (c) Sudoku solving

Figure 1.1.: Examples of constraints in these applications are: adjacent countries are drawn in different colors; that a resistor obey Ohm's law; or that a game of Sudoku may only be solved according to the rules.

While some problems can be expressed elegantly, constraint-based programmers must be aware of the limits of available solvers. One might argue that the imperative paradigm is more popular than constraint-based programming, because imperative programming works "just well enough" across a wide variety of problem domains. Constraints, on the other hand, work very well for specific kinds of problems, but then hardly at all for many other problems. An imperative program may become large and hard to understand, but the code can still be functional. When a constraint program becomes large, on the other hand, it can quickly approach the limits of current solving algorithms to produce a solution in any reasonable amount of time or at all.

Following the observation that purely constraint-based systems are unfeasible, but constraints do offer significant gains in some domains, "compound" paradigms such such as Constraint-Logic Programming (CLP) [69] or Constraint-Imperative Programming (CIP) [47] were proposed as attempts to combine the advantages of constraint programming with other general-purpose paradigms. There have been numerous attempts to implement these paradigms, for example, embedded in object-oriented programming languages. Some of these languages provide a constraint solving layer on top of an existing programming language [23, 71, 98], others integrate constraints directly into the underlying programming language by adding abstraction mechanisms and syntax for constraints [80, 53, 58].

Figure 1.1 shows applications in which a combination of imperative and constraint programming is particularly useful. For example, creating an imperative implementation of a Sudoku game (Figure 1.1c) works well for the graphical parts and user interaction, that is, for describing what should be displayed and what should happen when the user provides input. But an imperative algorithm for even a naive Sudoku solver in object-oriented code still requires dozens of lines of code. Using constraints, however, we can just describe the desired state that a valid Sudoku should have:

```
sudoku.cells().all_satisfy { |cell| 1 <= cell && cell <= 9 }
sudoku.rows().all_satisfy { |row| row.all_different() }
sudoku.columns().all_satisfy { |column| column.all_different() }
sudoku.boxes().all_satisfy { |box| box.all_different() }
```

These predicates assert that all cells must have numbers between 1 and 9 and that all rows, columns, and 3x3 boxes in the Sudoku must not contain duplicate numbers. A constraint solver can now take care of satisfying these constraints, and solve the Sudoku in the process. This frees the developer from writing or maintaining the code for the solving itself.

1.1. Challenges

Despite their advantages, languages that integrate constraints with imperative structures have failed to establish constraint solving as an integral part of imperative programs. Popular use cases for constraints, such as iOS and Macintosh auto-layout, planning and optimization, or model checking, still keep the constraints completely separate from the imperative code. We believe this is in part due to the inherent non-determinism in many constraint solving approaches, the lack of integration with existing imperative abstractions, the inexperience of the programmers with the features and limits of constraint solvers, as well as performance considerations. In this work, we look at these challenges and propose ways to address them.

First, a goal of this work is to define general properties for constraints that limit non-determinism and surprising solutions, regardless of the used solver. Many problems have multiple valid solutions, some more desirable than others. It is often unclear or only implicitly understood what criteria should be applied to choose a solution. Prior work on constraint languages sometimes included support for very powerful solver features that in practice were only rarely used, but that introduced non-determinism or allowed the solver to come up with unlikely and surprising solutions. A programmer had to understand the capabilities of each solver to know how to restrict what that solver could and could not choose as a solution.

Second, another goal of this work is to unify the abstraction mechanisms in a way that respects encapsulation and makes code re-use easier in semantically clean way. 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 CIP languages requires programmers to essentially learn a language with two very different parts. Abstraction mechanisms in the object-oriented parts — 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. Combining constraint code and imperative code syntactically can thus lead to more confusion about their differences than simply keeping constraints separated as a DSL.

A third goal of this work is to allow multiple solvers to cooperate, thus making efficient constraints solving available in a wider range of practical applications. It is in general 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 [121]. 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. This is apparent in many popular solvers: the Cassowary solver [3] can efficiently solve multi-way linear equations on floats using the simplex method; Z3 [90] is an satisfiability modulo theories (SMT) solver that can numerically solve for reals, integers, booleans, as well as record data types; Kodkod [118], Z3, and Ilog [63, 102] can enumerate solutions; and DeltaBlue [48] can solve multi-way 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., [62, 91]). These restrictions illustrate that, while constraint solving can be a powerful aid in programming, it should not be confused with general problem solving. Constraint solvers are specialized to address fairly trivial problems efficiently, so

that the developer can focus on other problems. Still, it is possible to expand the usefulness of constraint solvers through combination.

Finally, a fourth goal to make this work more usable in practice is to improve the performance of the solving process in general, and automatically make use solver specific features to improve performance, such as incremental resolving or re-ordering of constraint declarations. Prior work on CIP languages suffered from performance problems when accessing variables that had constraints on them. Additionally, although solving strategies such as incremental re-solving [48] and re-ordering of constraint declarations can improve performance of some solvers, these optimizations have to be applied explicitly and are specific to particular solvers. 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.

These goals — limiting non-determinism, integrating abstractions used for both paradigms, combining multiple solving strategies, and automatically optimizing solver performance — present a number of challenges. In this work, we set out to address some of these challenges as described below. To do so, we present a novel approach, which we call Object-Constraint Programming (OCP), to combine object-oriented code with constraint-programming in a way that deeply integrates constraint solving with the object-oriented paradigm and its abstractions.

Constraint Solvers are Non-deterministic "Magic"

A challenge for many constraint systems has been to not only find solutions, but *good* solutions, according to some optimization function. Constraint hierarchies [9, 11, 61, 119] and more recent work on optimizing SMT solver techniques [6, 17, 95, 113] provide the foundations to define optimization criteria for solvers, if these criteria are expressed appropriately.

In practice, humans use a lot of implicit criteria when searching for solutions to problems in a specific context. These criteria are often not only implicit but are in part not even consciously known, since solutions that do not satisfy them are not even consciously considered "valid". Thus, when the problem is put to the computer, many implicit constraints are omitted, preventing good optimization and 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, purely declarative programs can easily be made to scale to multiple threads of execution [111]), if something does not work as intended the gap between actual execution and user code is vast.

Type checkers and type inference system are special forms of constraint solving systems, as are model checkers that can, for example, compare object-oriented designs against the execution of their implementation. These system can infer many interesting properties and notify the programmer when an assumption cannot be proven. Approaches such as acceptability oriented computing [21] or input fuzzing [14] go one step further to not only check but also attempt to correct execution states of programs when assumptions fail to continue running.

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 values or also the types of variables; or even the semantics of the language? Is the identity of objects important, or can the solver substitute or conjure up new objects to satisfy its constraints? Which constraints

should lead to a run-time error when they are violated? Which should be fixed? Which can be safely ignored?

A design that attempts to make constraint solving available in the context of imperative languages must address these questions. It also needs to provide clear guidance to developers about what can be expected from the constraint solver, and needs to be explicit when it cannot satisfy some constraints and describe why.

Object-oriented Imperative Programming is the Norm

Considering the most popular languages according to language popularity indices such as Tiobe[1] or GitHut[2], imperative, object-oriented languages have 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.

In this work we aim towards a deeper integration of these two paradigms, to enable constraints in a wide variety of existing object-oriented (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 and syntax 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 phases. The first optimizes the solver's internal data structures

[1] `http://www.tiobe.com`, accessed December 11, 2015

[2] `http://githut.info/`, accessed December 11, 2015

to a particular problem, and the second actually solves. If only a subset of the variables is known to change frequently, the first phase can be optimized towards solving these, and the second phase becomes faster as a result. 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, 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 a 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 may thus deter from using a constraint solver in the first place. Languages such as Kaleidoscope [80] or Turtle [53], which integrate the constraints into the language and allow any variable to be used in constraints need more complex data structures and are thus further removed from a machine friendly representation that the JIT can map to the stack. Just like a modern JIT automatically lowers the representation of frequently accessed variables and, for example, stores them in registers of the executing CPU, a constraint solver must optimize its internal data structures to allow reading and re-solving for variables that change most frequently.

1.2. Contributions

In this thesis, we make the following contributions towards integrating constraints with imperative, object-oriented languages.

1.2.1. 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 a 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.

1.2.2. Linguistic and Meta-Level Symbiosis with Object-Oriented Host Languages

Our syntax and semantics are true extensions and thus backwards compatible to 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 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.

1.2.3. An Architecture for Adding and Using Multiple Cooperating Solvers

In addition to the goals of a clean integration, a semantics that is close to an object-oriented language, and good performance, a useful OCP language requires sufficiently powerful constraint solving capabilities. It is often infeasible to provide a single solving algorithm that works well across a variety of types and operations that occur in different application domains, and users of constraint solvers usually have to understand the limitations of the available solving strategies to use them effectively.

As part of this work we present a design and implementation for *cooperating constraint solvers* and *automatic solver selection*. Since most constraint solvers are specialized to work well for only a small number of aspects of a problem, a combination of different solvers may be more appropriate that work together to solve the entire problem. Babelsberg's architecture provides this capability, in a way that supports hard and soft constraints, automatically assigning constraints to different solvers, and fast incremental re-solving of such constraints.

To complement our architecture for cooperating solvers, we also present *automatic edit constraints* as an example for automatically using solver-specific optimizations if these are available. Edit constraints are a feature of select constraint solvers that can significantly improve the performance of constraint solving. However, using them requires developers to know about them, understand where they are useful, and to adapt the source code to create edit constraints. Consequently, they cannot be be used if the concrete solver is automatically selected from a group of solvers where not all support edit constraints. Thus, our design of automatic edit constraints shows how heuristics can complement our cooperating solvers design, to apply solver specific optimizations to improve the performance of the solving process.

Finally, our design also makes it straightforward to add new solvers, and it does not privilege the solvers provided with the basic implementation, so that user code can add specialized solvers that can then integrate with our architecture to solve more complex problems involving different types of constraints. Our architecture works with and without VM support and we demonstrate that it can address a variety of problems without the users learning specifically about the solving strategies involved.

1.2.4. Implementation Techniques and 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 Lively Kernel 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.

Regardless of the used implementation technique, we present a mechanism to automatically detect frequently changed variables and optimize the internal structures of constraint solvers to use incremental re-solving when these variables are assigned to. Our optimization provides a speedup of up to two orders of magnitude, and is completely transparent for the user.

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 Squeak and the Lively Kernel. We also show non-interactive, server-style applications and compare how constraint use differs between those two types of applications.

1.3. Outline

The rest of this thesis is structured as follows. Chapter 2 concludes the introductory part and presents and overview of the state of the art in three areas relevant to this work: First, we discuss common techniques in constraint solvers that are used to make interaction with mutable, imperative state easier. Second, we present how current approaches enable the use of constraints in imperative languages. Third, we give a general overview of different constraint solving techniques and the theory of satisfying hierarchies of competing constraints.

In Part II we present the design and semantics of Object-Constraint Programming. In that part, Chapter 3 first introduce the guiding principles and goals of our design using examples of various cases of how constraints and imperative structures interact. We start by explaining how constraints interact with primitive imperative structures and types, and incrementally build our design to integrate with a full object-oriented language, including methods, object identity, and higher-order functions. Afterwards, Chapter 4 formalizes our design into a big-step operational semantics. The formalization follows the same incremental steps of the design, and is meant to clarify ambiguity

and serve as a guide to language implementers. Finally, Chapter 5 present our approach for using multiple, cooperating constraint solvers to tackle problems that are too hard for any one solver, how the set of constraint solvers can be extended by users of an OCP language, and how the system supports users when choosing between solvers.

Part III describes the implementation strategies that were used in our three prototype implementations of OCP. In particular, we highlight pros and cons for implementing OCP as a VM extension versus a user-level library. In Chapter 6 we describe how constraint expressions are converted into constraints that can be passed to actual solvers, how these solvers are selected, and how operations and methods are transformed. In Chapter 7 we show how state mutation is intercepted to trigger constraint solving as necessary, and highlight some optimizations that we have made in practical implementations for performance reasons. Finally, Chapter 8 presents benchmark results to a) evaluate the performance overhead of our prototypes in comparison to unmodified, object-oriented VMs, b) compare our languages to other related languages that provide constraint solving in a general purpose programming language, and c) show the effect of using edit constraints and incremental constraint solving to improve the performance of some solvers.

Part IV gives an overview of our practical experience with OCP. Chapter 9 explains some coding patterns and idioms that we have found to work well in various applications, Chapter 10 describes a practical proposal for scoping the effect of constraints using Context-oriented Programming, and Chapter 11 presents tools that we have built to inspect, debug, and experiment with constraints in interactive applications.

Finally Part V discusses related and future work and presents our conclusions.

2. State of the Art of Constraint Solving and Constraints in General Purpose Programming

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 addresses 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 desired states — 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 zero.

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 [80].

Read-only Annotations We can remove the first case of non-determinism from the above example using the concept of read-only annotations [11]. 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, Freeman-Benson, and Wilson present a declarative specification of read-only annotations, adapted from formalizations of read-only annotations in committed-choice logic languages [84] — 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 zero. 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. Lopez, Freeman-Benson, and Borning 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 2.4.) 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* [79], 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 [86]. 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.)

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 [12], the *Mac OS X* [107] layout specification language, and the Python GUI framework *Enaml* [27]. This approach allows programmers to specify constraints and avoid boilerplate code to trigger constraint solving and has found widespread adoption, particularly through the Mac OS X layout system. However, these approaches require the developer to use an additional language when programming the system. The following is an example of an *Enaml* specification for our problem:

```
enameldef Main(Window):
   Container:
      constraints = [
         # the rectangle area is called contents in enamel
         contents_top >= 0, contents_left >= 0,
         (contents_bottom - contents_top) *
           (contents_right - contents_left) >= 100
      ]
```

Data-Flow and Functional Reactive Programming Some imperative 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 Scratch [103], the Lively Kernel/Webwerkstatt [66], or KScript [96]. (Of these, only ThingLab includes a planner for propagation that breaks cycles automatically. In the other systems the programmer has to break cycles explicitly.) The following listing shows an example of one-way data-flow in the Lively Kernel/Webwerkstatt:

```
connect(Window, "origin", Window, "origin",
      function(origin, prevOrigin) {
         if (!this.isVisible()) return prevOrigin;
         else return origin;
      })
connect(Window, "extent", Window, "extent",
      function(extent, prevExtent) {
         if (this.area() < 100) return prevExtent;
         else return extent;
      })
```

These 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. Here, the power of the solving is limited to uni-directional constraints, but the dependencies are clear and understandable.

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 [45] and its various extensions [80]. Related languages include Siri [59], Turtle [53], and SOUL [23]. Our constraints can be expressed using Kaleidoscope as follows:

```
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 `visibility` 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.

⁂

We think that these state of the art approaches result in trade-offs between constraint-oriented and object-oriented programming such that for some problems the problems of combining the two outweigh the possible benefits of using a constraint solver. In particular:

- Prior approaches violate either object encapsulation or the uniform access principle by defining constraints only on state or only on special constraint behavior definitions. This leads to problems as programs grow and implementations of objects change, as the constraints have to be kept up-to-date regarding the internal state of the domain objects.
- They use special types that tie objects to the domains of constraint solvers. This means that objects have to be mapped to those types either through transformations that have to be triggered whenever the program state changes or the objects have to be built entirely from those types.
- They provide no mechanism for programmers to add solvers for both finite and infinite domains.

Instead, in this work we propose an integration of constraint-solvers with OO languages that we call Object-Constraint Programming (OCP) to emphasize the focus on keeping OO programming as the primary paradigm. OCP keeps desirable properties from other approaches and is, for the most part, a continuation of CIP as originally presented in [45]. Table 2.1 shows these properties as presented above.

Table 2.1.: Comparison of related approaches

	Libraries	DSL	DataFlow/FRP	CIP
Constraints on methods				(✔)
Unified State		✔	✔	✔
Automatic Solving		✔	✔	✔
Linguistic Symbiosis			✔	✔
Extensible Solvers	✔		✔	
Bi-directional Constraints	✔	✔		✔

The goal of the work presented here is provide tighter integration of constraints with object-oriented languages. Using such an Object-Constraint Programming language, we should be able to express the above problem using the methods already defined for the Rectangle class:

```
rect = Rectangle.new
always { rect.area == 100 }
always { rect.visible? }
```

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.

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 One of the earliest constraint solving algorithms found in literature is local propagation. DeltaBlue [48] is a local propagation solver with support for acyclic constraints that

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

uses user-defined functions to propagate values. Local propagation is one of the simplest techniques to solve constraints, but it is restricted in that it only handles equalities of variables in a general way. For more complex relations, local propagation solvers require the user 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 once a solving plan has been created (which has to happen once per variable, and is in exponential time). 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 only a changes, DeltaBlue cannot decide whether to update b or c (or both) in response. An extension of DeltaBlue, SkyBlue [109], 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 [3] 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 [107] 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 (for 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 first transforms the constraint system 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 $x = 20 \wedge y = 25$.

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 [115] 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 Sketchpad'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 ∂, 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 by dividing the difference of the errors by our tweaking value. We subtract from x the original error divided by the coefficient and get 4.5 as new value for x. Since the error is now 0, we are finished, otherwise we iterate until we either converge to an error below some small or we give up.

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 oscillating. 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 [97] 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 [98] 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* [120], which reduces the domains of variables before and during the enumeration of values [92, 101]. 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 [90] 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 [6]. 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)) [20] 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 [6] or strings [122]. This makes Z3 useful for a wide-variety of problems that involve multiple types with required and non-required constraints.

⁂

In summary, even though powerful constraint solvers have been around for more than 50 years, relatively few systems have been built. Many constraint solvers still derive from fundamental ideas that were already present in the Sketchpad system [115]. Furthermore, the constraint solvers that have been built tend to be limited to specific application domains, making it hard to choose the right solver given a specific problem. Table 2.2 gives an overview of the presented types of solvers and their properties relevant for this work.

Table 2.2.: Overview of relevant constraint solving techniques

	Types	Functions	Multiple Outputs	Hierarchies	Optimal Solutions	Solution Precision	Performance
Local Propagation	Arbitrary	User-defined		✔	✔	=	++
Simplex	Reals	Linear	✔	✔	✔	=	+
Relaxation	Reals	Continuous	✔			≈	+/-
Backtracking	Finite Sets	Discrete	✔	✔	✔	=	--
SMT	Finite Sets, Integers, Reals	Discrete and Continuous	✔	✔	✔	=	+/--

2.4. 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 [11].

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 [3]. In Babelsberg, we use the priorities `required`, `high`, `medium`, and `low`.

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

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

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, Freeman-Benson, and Wilson [11] 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.

In general, a constraint might also consist not only of a single (in-)equality, but also conjunctions, disjunctions, and negations of atomic constraints. Only some solvers, such as Z3, can accommodate disjunctions and explicit conjunctions of constraints. For DeltaBlue and Cassowary, conjunctions of constraints are implicitly specified by feeding multiple constraints to the solver, while disjunctions are not allowed. For a conjunction or disjunction, if there is a priority, it applies to the entire constraint, not to components. Thus, this is legal:

$$\text{strong} \quad (x = 3 \vee x = 4)$$

but this is not allowed:

$$(\text{strong}\, x = 3) \vee (\text{weak}\, x = 4)$$

A more formal overview of constraint hierarchies is given in [11], and a summary of the theory that is relevant to our discussion here was published in [33]. For convenience, the latter is reproduced in Appendix C.

2.5. Good Performance in Practical Applications

We have discussed how constraint solvers are useful to find solutions to declaratively specified problems. However, especially when constraints interact with imperative code in graphical applications, the system state may change frequently as imperative code manipulates variables or adds and removes constraints between them. The system must be able to react to changes to maintain or re-satisfy constraints quickly, without reducing responsiveness to user input.

Kaleidoscope provides a declarative semantics for assignment, type declaration, and subclassing, where all of these are handled by a solver. However, this declarative semantics, while clean, only models the ordinary object-oriented semantics for consumption by the solver, and thus does not

add to the expressiveness of these constructs. Furthermore, this declarative semantics is also used if no actual constraints are in the program. In OCP, we decided to only use a 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.

If the constraint system is in a satisfied state, and most variables that are being manipulated are not referenced from constraints, the solver only runs for those assignments that would cause constraints to become invalidated. However, in interactive applications consistent performance is more important than performance that may be better on average, but where apparently similar manipulations sometimes are very fast and at other times noticeably slow [16]. Thus, even when the solver is run, the performance impact must be small enough so it will not noticeably affect the responsiveness of the system.

Incremental Re-Satisfaction Not all constraint solvers are developed to provide good performance for interactive applications [48], being used primarily for proofing or model finding [90]. However, of the ones that are used interactively, some solvers have explicit support for re-solving in case of state changes with good performance. To that end, Freeman-Benson and Maloney introduce the notion of *edit constraints* and incremental re-satisfaction. Their work is based on the observation that user input is usually restricted to modify only small parts of the constraint graph at a time, for example modifying 2D coordinates when moving the mouse or modifying a string by entering one character at a time.

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.

A disadvantage that arises from having a special interface for incremental re-solving is that it violates the uniform access principle. Changing a variable is now possible either through the edit constraint interface, or by adding and, after solving, immediately retracting a required equality. Unfortunately, at least for Cassowary, preparing the solver for fast incremental re-solving of a few variables requires some re-organization of the tableau, and solving for other variables also becomes slower as a result. This requires developers to anticipate whether a variable will only change more or less often, and use the appropriate interface if they want to achieve the best performance.

Summary

Developers trying to make use of constraint solving in stateful, imperative applications face trade-offs that influence the choice of how to interface with solvers and which solvers to use.

First, the spectrum of solver interfaces ranges from specialized DSLs for GUI applications, over general constraint DSLs such as the standardized SMT-LIB language [4], object-oriented libraries, to full language-level integration in Functional Reactive Programming (FRP) and CIP languages. The trade-offs here lie in how easy it is to re-use abstractions and integrate with existing imperative code versus how well the constraint code and thus its interactions with the system state is separated from the rest of the application. A shortcoming of all of these state the art approaches is that they do not integrate well with object-oriented concepts of encapsulation and behavioral abstraction.

Second, the solvers that are available vary wildly in their capabilities, performance, precision, and the potential to find surprising solutions. Features such as constraint hierarchies, stay and identity constraints, read-only annotations, and incremental resolving which provide more control over the solving process and performance are not universally available. They must also be applied explicitly, but there is little guidance or even principled rules of when they are needed. This problem is exacerbated in systems that attempt to integrate constraints at a language-level, because the requirements of concrete applications influence how these features should be used, while at the same time encapsulation and abstraction may make it hard to determine what features are needed in the first place.

The work presented here is motivated by the belief that there is room for a language design with tighter integration of constraints and object-oriented features such as methods and inheritance. This design should allow using multiple solvers and solver-specific features in a principled manner without breaking encapsulation and without requiring intimate knowledge of the different solving procedures that are available in the system.

Part II.

Object-Constraint Programming: Design and Semantics

3. Design of Object-Constraint Programming Languages

An overarching design goal is that in the absence of constraints, an Object-Constraint Programming (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.

Experience with integrating constraints into object-oriented languages [45, 47, 46, 79] has shown that a number of important problems involve constraints including mutable objects, method activation, inheritance, object identity, and finite collections of objects. These kinds of constraints presents a number of challenges and possibly confusing issues, such as the potential for finding surprising solutions and non-determinism. If a constraint references a field that does not exist on an object, should the solver be allowed to add it? If a constraint requires two variables to refer to the same object, which variable should be changed? Which methods can be used in constraints? Can they only be used in the "forward" direction, to compute a result, or can the solver also use them to define multi-directional constraints? For the imperative programmer with little prior experience in constraint programming, incrementally adding constraints was thus often difficult, as the possible effects of a constraint could not be easily determined. As part of our design, we have proposed a set of design principles to avoid such surprises, as well as restrictions on OCP languages to enforce these principles [38]:

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

methods called as part of that expression also follow this restriction. The remaining methods can be used just in the "forward" direction, in order to produce a result value which can participate within constraints.

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. Furthermore, when expressing constraints over complex objects, a local propagation solver such as DeltaBlue may often produce better results.

Another goal for our design is thus to abstract from the concrete implementation of the used solvers. As long as a solver supports constraint hierarchies over some primitive types that are also in the host language, and we can encode uninterpreted symbols and functions for it in some way, our design should be usable with that solver. The solver should furthermore be sound (but need not be complete), and find a single best solution — if there are multiple solutions, the solver is free to pick any one of them. Providing answers rather than solutions, i.e., results such as $10 \leq x \leq 20$ rather than a single value for x, and backtracking among multiple answers, as available in for example constraint logic programming [68], is not part of this design.

Our design should enforce our principles regardless of how the solver is implemented. 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 variables that have primitive type are mapped to the solver, how the solving process interacts with these variables' 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 primitive types, 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.

We will illustrate our design decisions using small example programs and their associated constraints. Programs are written in fixed pitch font, the constraints that are handed to the solver in math font.

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. We call this language Babelsberg/PrimitiveTypes. In this first step, we omit user-defined, structured classes or methods, 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 imperative with mostly standard evaluation rules. The language includes operations on the primitive values, mutable variables, and standard imperative control

structures such as branches and loops. Operations on the variables are executed primitively as part of the language, rather than as message sends. It is straightforward to extend this language with other primitive types, provided that they all are atomic — we do not have recursive types or types that define values that hold other values (such as records, arrays, or sets). Primitive types as we regard them map to types natively supported by the solver. They are not, however, necessarily primitive in the host language. For example, in languages like Ruby or Smalltalk, integers, reals, and booleans are all proper objects.

3.1.1. Declaring Constraints

The first extension over a basic imperative 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 will be 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.

3.1.2. Translating and Solving Constraints

The evaluation model for Babelsberg/PrimitiveTypes is mostly standard for ordinary expressions and statements. In contrast, a statement that adds a constraint starts with a duration, namely `always` or `once`. The expression following the duration is taken added to the *constraint store* unevaluated. The only adjustment in the evaluation model for ordinary statements is that for an assignment statement, we evaluate the expression on the right hand side of the assignment, constrain the variable on the left hand side of the assignment to be equal to the result using a `once` constraint, and then turn all the constraints over to the solver. Doing this ensures that assignment interacts correctly with other constraints.

Here is a simple example:

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

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

$$\text{required} \quad x = 3$$

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

$$\begin{aligned} \text{weak} \quad & x = 3 \\ \text{required} \quad & x = 4 \end{aligned}$$

The weak $x = 3$ constraint is the *stay constraint* that `x` retain its previous value, while the required $x = 4$ constraint comes from the second assignment. These stay constraints are generated implicitly by the language. They contribute to our goal of not surprising the programmer. In imperative languages, programmers expect their variables to retain their values, unless some assignment operation changes them. For example, in interactive graphical applications, when moving one part of a

constrained figure, the user generally expects other parts to remain where they are unless there is some reason for them to change to satisfy the constraints. As described in Section 2.1, this desire is commonly expressed using stay constraints. As mentioned above, we assume in the design that the solver supports low-priority stay constraints, although in practice not all do. (We describe later how we deal with this in the practical implementations.)

After the second statement, the solver finds the solution $x = 4$, resulting in a new environment with `x` $\mapsto$ `4`. The third statement adds the `always` constraint to the constraint store. So after that statement we have the following constraints:

$$\begin{array}{rl} \text{weak} & x = 4 \\ \text{required} & x \geq 10 \end{array}$$

If we use a metric comparator such as weighted-sum-better or least-squares-better, we get the solution $x = 10$, since this minimizes the error for the weak constraint. If we use locally-predicate-better, then every $x \in [10, \infty)$ is a solution, and the system is free to select any of them. (However, as noted in Section 2.4, typically we would not use locally-predicate-better if we have inequalities.)

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 that equates the variable on the left hand side to the 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.

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

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

After evaluating the first two statements and solving the resulting constraints, the environment has the binding `x` $\mapsto$ `3`, `y` $\mapsto$ `0`. The third statement causes the constraint `always y = x+100` to be added to the constraint store. We then hand the following constraints to the solver to find values for `x` and `y`:

$$\begin{array}{rl} \text{weak} & x = 3 \\ \text{weak} & y = 0 \\ \text{required} & y = x + 100 \end{array}$$

This has multiple possible solutions, and the solver can select any one of them. Suppose it picks `x` $\mapsto$ 3, `y` $\mapsto$ 103. After evaluating the next statement, we have the following constraints:

$$\begin{array}{rl} \text{weak} & x = 3 \\ \text{weak} & y = 103 \\ \text{required} & y = x + 100 \\ \text{required} & x = 5 \end{array}$$

The first two constraints are the weak stays on x and y, the third comes from the always constraint in the constraint store, and the fourth comes from the assignment `x:=x+2` (where we evaluated `x+2` in the old environment to get 5). After solving these constraints, we have x ↦ 5, y ↦ 105.

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

```
always x=10;
```

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

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

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

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

As an example of unsatisfiable constraints, consider:

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

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

$$\begin{array}{rl} \text{weak} & x = 5 \\ \text{required} & x \leq 10 \end{array}$$

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

$$\begin{array}{rl} \text{weak} & x = 5 \\ \text{required} & x \leq 10 \\ \text{required} & x = 20 \end{array}$$

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

An interesting aspect of this semantics is how we model assignment as a `once` constraint between a variable and an evaluated value on the right-hand side. We derive this model from the Kaleidoscope system. Kaleidoscope initially modeled assignment as constraints between variables at different times in what is called a *refinement model* of constraints. This model, adapted from Lucid [2], adds some complexity to the semantics. Ordinary variables are represented as a stream of "pellucid variables," each holding a value at a different time, and we have to ensure that values

can only flow forward in time, and with the details of when to advance the time and thus which values a given set of expressions can access. Later versions of Kaleidoscope switched instead to the *perturbation model*, which we also use here. In this model, variables are instead represented conventionally as holding a single value. Assignment *perturbs* the value of a variable, and then the constraints are then solved to adjust the values of variables so that they are re-satisfied if necessary.

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

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

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

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

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

3.1.4. Requirements for Constraint Expressions

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

1. Evaluating the expression that defines the constraint should return a boolean. (This can be checked statically or dynamically.)
2. The constraint expression should be free of side effects.[1]
3. The result of evaluating the block should be deterministic. For example, an expression whose value depended on which of two processes happened to complete first would not qualify. (This does not arise in the toy languages here, although we do need this restriction for a practical one.)

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

An intuitive reason for these restrictions is that there are two interpretations for each constraint at a particular step in the execution of a program in our design: first, saying the constraint is satisfied is equivalent to saying that the result of evaluating the constraint expression is `true`; and second, saying the constraint is satisfied is equivalent to saying that the generated constraint was satisfied by the solver. The behavior of an interpreter or compiler for an implementation of the Babelsberg design should match this interpretation. Thus, we need to be careful not to hand constraints to the solver that it may think are satisfiable, but which are, in the semantics of the imperative language, not so.

3.1.5. Control Structures

Babelsberg/PrimitiveTypes includes `if` and `while` control structures. These work in the usual way, and allow (for example) a variable to be incremented only if a test is satisfied, or an `always` constraint to be conditionally asserted. The test for an `if` statement is evaluated, and one or the other branch is taken — there is no notion of backtracking to try the other branch. Similarly, a `while` statement executes the body a fixed number of times — there is no possibility of backtracking to execute it a different number of times.

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

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

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

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

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

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

3.2. Constraints on Objects and Messages

While the previous section illustrates the fundamental interactions of constraints with variable assignments and their integration with control structures and the syntax of the language, this section will illustrate how this extends to a fully object-oriented language. For this next language, which we call Babelsberg/Objects, we will add mutable and immutable objects that reside on the heap or the stack, respectively. We also illustrate constraints over object identity, constraints on the results of messages sends and how these are handed to the solver, as well as the limitations as to which kinds of messages can be used in constraints.

3.2.1. Objects, Structure, and Identity

Babelsberg/Objects has ordinary objects that reside on the heap and have a unique identity. Allowing constraints over object identity is useful for example for defining data structures (for example a circular linked list) or when modeling aspects of the real world [79]. For example, it is clearly different if two train tickets are valid for the *same* or an *equal* train. Thus, we add identity constraints to the language (following Smalltalk syntax, written ==, in contrast to = for equality constraints). For two objects p and q, if p and q are identical, they must also be equal, but the converse is not necessarily true — if p and q are equal, they might or might not be identical.

Among our restrictions on constraint expressions in Section 3.1.4 was that they must be free of observable side-effects. In this language, we consider creating an object a side-effect. This is because object creation could potentially be observed using a meta-protocol, it modifies the heap, and newly created objects can be tested for identity, which would be equivalent to testing against a random number. Thus, new objects cannot be created as part of constraints.

Since ordinary objects cannot be created in constraint expressions, Babelsberg/Objects includes *value classes* as well as ordinary, full-featured, classes. Instances of value classes *can* be created in constraints. They are immutable after object creation[2], and object identity is not significant for them — an identity test performs a test for equality of structure and fields. However, value classes are more than simple record declarations, since they support methods and inheritance.

In a practical implementation of Babelsberg, ideally the host language will itself support value classes, so that we can use them directly. A number of existing languages have proposals or support for value classes, such as Java[3] or Scala[4]. If not, the implementation may relax some of the restrictions we have on creating ordinary objects in constraints, and instead enforce appropriate conventions about object modification and not testing the identity of objects that were created in constraints.

We will now show how our complete, object-oriented design encodes the principles from Chapter 3: Identity preservation, structure preservation, and identity/structural determinism.

3.2.2. Identity Preservation

In addition to the stay constraints that variables of primitive type stay equal to their primitive values, we now also automatically generate stay constraints on variables that store references. Thus, a variable continues to refer to the same object (and no just an equal one) unless reassigned. This requires the solver to support uninterpreted values — the stays ensure that the variables are equal to the (uninterpreted) references that they hold, not to the objects on the heap. In addition, we add stay constraints for each field of an object. Since all practical solvers find minimal solutions (rather than inventing variables that are not mentioned anywhere), these stay constraints ensure that the solver cannot remove fields of an object if not all have been explicitly mentioned in constraints.

Consider the following example:

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

The first line creates a new object with two fields on the heap. When evaluating the second line, these are the generated constraints:

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

[3] `http://openjdk.java.net/jeps/169`, accessed December 11, 2015

[4] `http://docs.scala-lang.org/sips/completed/value-classes.html`, accessed December 11, 2015

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

The reference r is an uninterpreted constant. We assume the solver understands uninterpreted function symbols. The uninterpreted function `H` in the second constraint above is used to model the heap's mapping from references to objects. The solver cannot change this mapping, only update the fields if a higher priority constraint requires it. Marking this constraint as required ensures that the solver's solution will always agree with what is on the heap — namely, that reference r points to an object that has *only* an x and a y field. We introduce variables for the values of these fields and add weak stays to these, so the solver can find solutions that update parts of objects.

The third line adds a constraint $a = p$, which from then onwards ensures that any assignment to either of those variables is reflected on the other.

The stay constraints above would be enough to ensure that the identity of `p` cannot change in this case. However, consider the following example:

```
p := new {x: 0};
a := new {x: 2};
always p.x > 0
```

Here, the desire may be that the system change `p.x` to be larger than zero. However, the resulting constraint system using only weak stay constraints would allow the solver to satisfy the constraint by making `p` point to the reference stored in `a`! This in itself may seem useful for some applications, but it is a slippery slope. There is inherent non-determinism in allowing such a change, because we cannot tell which reference `p` will point to afterwards. This becomes especially apparent if we changed the last constraint to `always p.x = 0 || p.x = 3`. Here, the system may even choose to change `p` to point to the second object *and* change that object's `x` field, depending on the comparator used.

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

```
p := new {x: 0};
a := new {x: 0};
always a == p
```

This identity constraint requires two variables to point to the same object. It is not clear whether the solution will require them both to point to the object stored in `p` or that stored in `a`. The solver could also satisfy the identity constraint by assigning both variables to yet a third object[5]. Adding a read-only annotation on one of the variables does not suffice, because that would prevent the system from re-satisfying the constraint if we later assign to the variable that is not read-only.

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

To address these problems, we employ a principle of *identity preservation*: newly asserted constraints cannot change which object a particular variable or field stores. When ordinary constraints are asserted, we make all stay constraints that equate variables and references be `required`. Second, we require all identity constraints are *already satisfied* at the time they are asserted — they are only tested at the time they are asserted, and not handed to the solver. Only during later assignments will the system re-satisfy the constraint deterministically. Third, to be able to enforce these properties, we do not allow identity constraints to be mixed with non-identity constraints. If either of these conditions is not met, we treat the constraint as unsatisfiable. While this design places a bit more burden on the programmer, we believe that the programmer effort is more than made up for by the strong guarantee provided by the language.

3.2.3. Structure Preservation

With objects and, in particular, references to their fields being allowed to appear in constraint expressions, situations can arise in which the solver could come up with an undesired or surprising solution. In this section, we will use `Point` and `Rectangle` as value classes. To distinguish them from ordinary classes, we will create instances just by supplying the arguments, e.g., `Point(10, 20)`, whereas instances of ordinary classes will be created using the `new` message, e.g., `new Window(...)`. Finally, to foreshadow the formal semantics, for simplicity we omit details about object creation, method lookup, and inheritance (which are completely standard and inherited from the host language).

The following example assumes that a `Window` has a `topLeft` field, which contains an instance of the value class `Point`:

```
a := new Window(Point(0, 0));
always a.topLeft = Point(10, 10)
```

If we translate these constraints using required stays on variables with references (as per Section 3.2.2), but with weak stay constraints for other variables, we get the following constraints:

$$\begin{array}{rl} \text{required} & a = r \\ \text{required} & \mathtt{H}(r) = \{topLeft : topLeft_r\} \\ \text{weak} & topLeft_r = \{x : 0, y : 0\} \\ \text{required} & \mathtt{H}(a).topLeft = \{x : 10, y : 10\} \end{array}$$

In this example, no structurally surprising solutions are possible — the solution is for the solver to replace `topLeft` with a new value class object that has the correct values for its `x` and `y` fields. However, suppose we had asked the solver to make the `topLeft` field of the window above be equal to a 3-d point: `always a.topLeft = 3DPoint(10, 10, 10)`. Should the solver be allowed to add a `z` field, so that `a.topLeft` suddenly has a different structure? After all, we have asked it to ensure that `a` be structurally equal to an object having such a field. While such behavior may seem desirable in this case, it can cause non-determinism. What if the constraint is of the following form:

```
always a.topLeft = 3DPoint(10, 10, 10) || a.topLeft = Point(20, 20)
```

Should the solver be allowed to change the structure of `a` depending on what works best with other constraints? If `a` currently has no fields at all, which ones should it gain?

Allowing such constraints, although it adds to the power of the system, would make programs using them harder to understand, as the structure of an object cannot be determined just by looking at the code. To avoid this, we employ a principle of *structure preservation*: the solution to a newly

asserted constraint will never change the structure of any objects, where we take *structure* to include the names and number of an object's fields (and thus, implicitly also its size), recursively. We enforce this principle by employing a form of *structural type-checking* at run time, just before sending constraints to the solver. Our checks ensure that all structural requirements of the given constraints are already satisfied in the current environment and heap. For example, before solving the constraint on the `topLeft` of our window, a Babelsberg language must check that the field already all fields required to make it equal to a point, and will raise an exception if that is not the case.

Furthermore, constraints on parts of a structure are translated to the solver in a way that disallows changing any but the last part of the structure that is named in the constraint. Consider the following constraint for a tree of instances of value classes:

```
a := {b: {c: {d: 5}}};
always a.b.c.d = 10
```

The desired outcome is clearly that the solver makes the `d` field of the object `c` equal to `10`. However, a valid solution might be to set `c` to a different object that has the right `d` value, or even change `b` or `a`. These solutions are undesirable, and we prohibit it by translating this expression to the following constraints:

$$\begin{array}{rl} \text{required} & a = b : b_v \\ \text{required} & b_v = c : c_v \\ \text{required} & c_v = d : d_v \\ \text{weak} & d_v = 5 \\ \text{required} & a.b.c.d = 10 \end{array}$$

Value Classes as Sugar Value classes are in fact not necessary for Babelsberg — there is a simple transformation that can be used to remove them. However, they are useful in practice, since using them can make Babelsberg/Objects programs much clearer and simplifies an implementation of Babelsberg that follows our restrictions. However, showing how they can be eliminated is useful to illustrate how our *structural type checking* avoids verbosity in the constraints.

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

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

We can rewrite this as:

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

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

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

is rewritten as:

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

The read-only annotations are necessary since we do not want to satisfy the constraint by changing `d`.

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

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

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

At the end of this program `a = Point(30, 20)`, but `b = Point(10, 20)`. However, if we do the same thing with mutable objects:

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

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

3.2.4. Deterministic Updates to Structure and Identity

Of course, imperative programs do need to sometimes update the structures of objects and change which objects are stored in particular variables and fields. In Babelsberg we must therefore allow such updates. The challenge, then, is to provide this expressiveness while at the same time avoiding non-determinism in the structures or identities of objects.

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

As described in above, constraint solving is not allowed to modify references stored in variables. To allow this during assignment, we therefore introduce a two-phase approach to constraint solving as part of an assignment statement. In the first phase, all identity constraints are re-established as described above. In this phase, the stay constraints on identities and value class structures are weak,

and the solver may update the structure of objects and change which objects are stored in particular variables and fields, but it will do so deterministically. In the second phase, all other constraints are solved in the context of this updated environment and heap, using the restrictions described earlier, and with `required` stay constraints on the structures and references stored in variables.

Assigning Mutable Objects So far we have described how assignment to a variable is handled with primitive objects on the right hand side. We have not addressed the case of mutable objects with substructure on the right-hand side. Consider the following midpoint line:

```
line := {start: new MutablePoint(0,0),
        end: new MutablePoint(1,1),
        midpoint: new MutablePoint(0,0)};
always line.midpoint = line.start.addPoint(line.end).scaleBy(0.5);
line.midpoint := new MutablePoint(0, 0)
```

There are two ways to look at such an assignment: a) the assignment asserts equality between `midpoint` and `MutablePoint(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 [78], 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. One practical solution 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, but if it cannot use the new midpoint, it can also try and fix the parts of the right-hand side.

3.2.5. Control Structures and Methods

The full Babelsberg/Objects design still has the simple `if` and `while` control structures that were introduced for Babelsberg/PrimitiveTypes. In addition, however, Babelsberg/Objects includes methods. Following our overarching design goal of having a standard object-oriented language in the absence of constraints, in imperative execution mode, methods are standard. Suppose we are evaluating `x.m(a, b)`. We first look in `x`'s class for a method with selector `m`, then its superclass, and so on up the superclass chain. If no such method is found, it is an error. [6] Otherwise we evaluate the body of the method in a new scope, and return the result. Arguments are passed by value (with pointer semantics), as in most standard object-oriented languages. For instances of primitive types and value classes, the argument can also be copied (since we cannot tell the difference between sharing and copying in this case).

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

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

The constraint here is on the result of sending the message `center` to `r`, then sending the `x` message to that, and finally sending the = message to the result. Depending on how the rectangle

[6] Note that we could have chosen to give the system additional power to invent a new method or convert an object to an instance of a different class to satisfy a method lookup. However, similar behavior has not been useful in practice in the earlier Kaleidoscope [45] and BackTalk [98] systems, and would also go against our goal of keeping close to a standard object-oriented language.

is stored, the rectangle's center may well be computed in the `center` method rather than simply being looked up; and even `x` might be computed rather than being stored, for example if the point is stored in polar coordinates.

Constraint expressions in this design are not simply executed, but instead interpreted using an additional evaluation mode we call *constraint construction mode (*CCM*)*. While in the simple Babelsberg/PrimitiveTypes design presented before we assumed that the solver primitives and its operations are essentially the same as those in the imperative language, constraint expressions on objects and messages are now translated into the language of the solver. We still assume the solver can deal with some primitive types, but it does not know about methods or inheritance. For simplicity, we will assume that the solver can understand uninterpreted symbols and records. We will show later that this is not strictly necessary to implement a practical OCP language, but it simplifies the description of the design. In this full design of Babelsberg, we treat all operations on objects such as `a + b` in an object-oriented fashion, so that this means "send the message + with the argument `b` to the object `a`," with the meaning of + in this expression (and in constraints such as `a + b = c`) depending on the class of `a`.

As with the previous languages, semantically, after every statement execution that could potentially invalidate constraints, the current set of active constraints is translated, solved, and the environment is updated with the solution (or the program halts with an exception). A practical implementation can optimize by caching the translated constraints and only invalidate the cache as necessary, or only solve a subset of the constraints.

Methods implemented as primitives in the host language (such as + on integers), are translated into the equivalent operations in the solver and the participating objects are unboxed. For a language like Java, which has true primitive values, the operations can still be translated directly, but primitive methods on boxed types like `Integer`, `Float`, `Boolean`, and so forth are translated into primitive operations on their unboxed equivalents. When we encounter a message in CCM that is not a primitive, we do a normal method lookup, transform, and inline the method body, and then continue to evaluate the inlined body in CCM.

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

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

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

After the program runs, `y` will be 20 and `x` will be 10. During the execution if the `always` statement, the constraint expression `y = x.double()` explodes into the following network of constraints:

$$\begin{array}{ll} \text{required} & \mathit{self} = x \\ \text{required} & y = 2 * \mathit{self} \end{array}$$

This conjunction of constraints works equally well for determining `x` or determining `y`, even though `double` is actually a message to `x`. However, not all methods can be used in this manner. There are a number of restrictions on methods to allow them to be used in this way, which we discuss

in more detail in Section 3.2.6. Briefly, if they have side effects the transformation fails. If they consist of more than just a single return statement they can only be used in the *forward direction* in constraints, i.e., they return value will be passed to the solver as a constant, and all their arguments are marked as being read-only for the solver. Creating an instance of an ordinary class is regarded as a side effect, so methods that can be invoked by constraint expressions can only create instances of value classes. If we we encounter a side-effect, the transformation fails. If we encounter a complex expression or statement that we cannot transform, we evaluate it normally and only use its result.

As an example for a constraint in which a method can only be used in the forward direction, suppose we have `always c = x.m(a, b)`, and method `m` has multiple statements so that it cannot be translated and passed to the solver. We will evaluate the method and pass the equality between its result and `c` to the solver, as well as required equalities for `x`, `a`, and `b` to keep their current values. This allows the system to find an assignment for `c` given values for `x`, `a`, and `b`, but not any other direction. If any other constraints are such that the correct solution involves finding a value for `b` given values for `c`, `x`, and `a`, a practical system will halt with an error that the constraints are too hard for it to solve, due to a method called from the constraint that cannot be inlined (or in the formal semantics, we get stuck). However, in contrast to all the previous examples of constraints that were too hard, the fact that they are too hard is a limitation of the transformations we use for methods called by constraints, rather than being a limitation of the solver.

On the other hand, if method `m` can be used multi-directionally, so that for example we can find a value for `b` given values for `x`, `a`, and `c`, then in the semantics the method is translated and inlined so that the resulting constraint expression can be turned over to the solver. This done by creating a scope and fresh variable bindings for method `m` and inlining the returned expression with respect to that environment. The parameters are constrained to be identical to the argument names used in the fresh scope for the method, and the receiver is constrained to be equal to `self` in the new scope.

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

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

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

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

Note that since `Point` is a value class, we can make an instance of it in the constraint expression itself (“`Point(10, 20)`”), and also in the `center` method, since we are doing point addition and division in that method.

The constraint is evaluated in the following way. The code for the rectangle’s `center` method is found via standard object-oriented method lookup. It is inlined to construct an equality constraint between a new point with `x=10` and `y=20` and the result of evaluating `(self.upper_left + self.lower_right) / 2`. Constructing the center point from the first expression, as well as looking up and inlining the `=` method on it, explodes into a network of simpler constraints, all required, that can then be handed to the solver in one conjunction. The table below

shows how the constraints are constructed, and in particular how the local names are translated into a global solver environment by making them unique.

```
                                                    // constraint for the receiver of the center method
required   self1 = {upper_left : {x : 2, y : 2}, lower_right : {x : 10, y : 10}}
                                                    // the point addition called from the center method
required   self2 = self1.upper_left
required   arg1 = self1.lower_right
                                                    // the point constructor called from the point addition method
required   x1 = self2.x + arg1.x
required   y1 = self2.y + arg1.y
                                                    // point division by scalar called from center method
required   self3 = {x : x1, y : y2}
required   arg2 = 2
                                                    // point constructor called from point division method
required   x2 = self3.x / arg2
required   y2 = self3.y / arg2
                                                    // point constructor for the static point
required   x3 = 10
required   y3 = 20
                                                    // and for the point equality
required     self4 = {x : x2, y : y2}
required     arg3 = {x : x3, y : y3}
required   self4.x = arg3.x ∧ self4.y = arg3.y
```

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

Now consider a similar example but using mutable rectangles and points on the heap, using the translation when using their constructors in constraints as defined in Section 3.2.3.

```
                                                    // stays for the heap
required     H(r) = {upper_left : p_ul, lower_right : p_lr}
required   H(r_ul) = {x : x1, y : y1}
required   H(r_lr) = {x : x2, y : y2}
required     p_ul = r_ul
required     p_lr = r_lr
required       x1 = 2
required       y1 = 2
required       x2 = 10
required       y2 = 10
```

$$
\begin{array}{ll}
 & \text{// the center method} \\
\text{required} & self_1 = r \\
 & \text{// the point addition called from the center method} \\
\text{required} & self_2 = \mathsf{H}(self_1).upper\_left \\
\text{required} & arg_1 = \mathsf{H}(self_1).lower\_right \\
 & \text{// the point constructor called from the point addition method} \\
\text{required} & \mathsf{H}(r_c) = \{x : x_3,\ y : y_3\} \\
\text{required} & self_3 = r_c \\
\text{required} & x_3 = \mathsf{H}(self_2).x + \mathsf{H}(arg_1).x \\
\text{required} & y_3 = \mathsf{H}(self_2).y + \mathsf{H}(arg_1).y \\
 & \text{// point division by scalar called from center method} \\
\text{required} & self_4 = self_3 \\
\text{required} & arg_2 = 2 \\
 & \text{// point constructor called from point division method} \\
\text{required} & \mathsf{H}(r_d) = \{x : x_4,\ y : y_4\} \\
\text{required} & self_5 = r_d \\
\text{required} & x_4 = \mathsf{H}(self_4).x \,/\, arg_2 \\
\text{required} & y_4 = \mathsf{H}(self_4).y \,/\, arg_2 \\
 & \text{// point constructor for the static point} \\
\text{required} & \mathsf{H}(r_s) = \{x : x_5,\ y : y_5\} \\
\text{required} & self_6 = r_s \\
\text{required} & x_5 = 10 \\
\text{required} & y_5 = 20 \\
 & \text{// and for the point equality} \\
\text{required} & self_7 = self_5 \\
\text{required} & arg_3 = self_6 \\
\text{required} & \mathsf{H}(self_7).x = \mathsf{H}(arg_3).x \wedge \mathsf{H}(self_7).y = \mathsf{H}(arg_3).y
\end{array}
$$

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

This example also illustrates another reason why we need to solve for object identity and type before values (cf. Section 3.2.4) — if `r` had been an instance of a class with a different implementation of the `center` method, it would have been inlined in a different way and created different constraints. For the same reason the receivers of messages sent in constraint expressions cannot be allowed to change their types, because this, too, may change which method should be executed in response to the message.[7]

[7] Note that Kaleidoscope did allow such changes to the receiver and could try different methods to satisfy the constraints. We decided that such behavior made the solving process very unpredictable and decided against it.

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

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

Now suppose we call `require_min_balance(a, 10)` on some account `a` (as an ordinary method invocation, *not* from a constraint expression). Thereafter the balance in the account `a` must be at least 10 (a persistent constraint). We use a `?` annotation on the variable `min` to make it read-only with respect to this constraint — otherwise the solver would be free to change either the account balance or the local variable `min`, hiding any effect of the constraint.

Note the consequences of calling this method with arguments passed by value (with pointer semantics):

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

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

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

After the second assignment, `a` is bound to a new account — but the minimum balance constraint is on the previous instance of `BankAccount` (which can be garbage collected since there are no references to it). However, in general implementing such methods requires that the semantics keep the local variables in such a method invocation as long as the constraints it creates are active and any one of the variables is still referenced from the outside.

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

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

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

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

As a common pattern, placing constraints inside methods allows programmers to express structural integrity constraints inside the constructors for objects, and thus ensure they are satisfied throughout the lifetime of the object. This next example ensures that a bank account is always initialized with a minimum balance (defaulting to zero):

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

3.2.6. Additional Restrictions on Constraint Expressions

Methods can of course have side effects, but such methods cannot be used in the expressions that define constraints. (This is one of the restrictions noted in Section 3.1.4 on requirements for constraint expressions.) This subsection describes some consequences of this restriction when we have objects, methods, and object identity. There are also some additional restrictions on methods called from constraints that are used in other than the forward direction, discussed below.

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

```
x := stack.pop()
```

This works, because the RHS of the assignment is evaluated first, and then we set up a `once` constraint using the value popped from the stack. (Thus the RHS of an assignment can have side effects, in contrast to `always` or `once` constraints.) In contrast, this fragment is not allowed, because we cannot repeatedly evaluate the constraint expression as a test:

```
always x == stack.pop()
```

Instances of Ordinary Classes As noted previously, we consider creating an instance of an ordinary class a side effect, so methods that can be invoked by constraint expressions can only create instances of value classes and not ordinary classes. As noted previously, a practical implementation might be able to allow cautious use of benign side effects in constraint expressions, including creating instances of ordinary classes. For example, constructing a temporary instance of an ordinary class that is garbage collected before it is visible outside the constraint should not be a problem. However, In the design and formal semantics, however, we simply disallow such side effects in constraint expressions.

Nested Constraints Creating a constraint is a kind of side effect, at least potentially, so another consequence of the prohibition on side effects in constraint expressions is that methods called from a constraint expression cannot themselves create other constraints, or call further methods that do so.

One issue is that adding more identical constraints is not an idempotent operation when using constraint hierarchies. The following program illustrates this:

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

x := 0;
y := 0;
```

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

Suppose we are using a least-squares solver. Every time we evaluate the expression `y=test(x)`, we add another medium constraint that `x = 5`. Since we are finding a least squares solution, this nudges `x` more toward `5` each time, diminishing the influence of the `x = 10` constraint. However, if we used a local comparator, re-adding the soft constraint would not change the resulting solution. Since we want our design to be independent of such details, we must disallow this use.

Another reason why nested constraints are disallowed is because the method that adds soft constraints might itself be invoked by a soft constraint — for example, suppose that in the above program we instead had `always weak b = test(a)`. What should the priority of the inner constraint be? We would need an algebra for combining priorities.

Finally, allowing nested constraints means we would have to solve eagerly rather than translating and inlining the expression and then solving all resulting constraints at once. Consider the following example:

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

a := 5;
always b = test(a)
```

The expected outcome is that the method `test` always returns a value larger than `10`. If we do not solve eagerly, however, `y = 11` and `a = x = 5` when we evaluate the `if` statement. Thus, when used in a constraint, this method could return a value smaller than `10` (in this case `5`).

Assignments Methods called by constraints can assign to variables. If there are assignments, the method can only be used in the forward direction in constraints that call it (i.e., to compute the result given the inputs). However, programmers must be careful with assignments that would be visible outside the method. A valid use-case for global assignment is caching and lazy initialization, but it would produce unpredictable results if, for example, a global counter were incremented and returned on each call.

3.3. Constraints on Collections of Objects

While it is not strictly necessary to include collections such as arrays or sets in Babelsberg, supporting collections is useful in a variety of applications that use constraint-based programming. In particular, finite domain problems such as those found in logic puzzles often lend themselves well constraint-solving techniques, while their graphical representation and user-interface is usually more readily expressed using imperative code. This combination makes them an ideal example of the kinds of applications we want to support with this design.

3.3.1. Constraints on Collection Predicates

Until now, we have expressed constraints that correlate single variables or fields. Suppose that arrays in our language have a `length` field and a number of indexed fields starting at `0`, and that we can access these fields using square bracket notation. Thus treat the length and fields of an array as

its structure, and use the same structural compatibility checks for the existence of array elements that we do for the existence of fields in a object. Thus, we would not, for example, grow an array to allow a constraint on its i^{th} element to be satisfied if it did not already have an i^{th} element. If we want to express that all fields in such an array should be pairwise different, we can do so in Babelsberg/Objects with a loop:

```
i := 0;
while i < array.length do (
  j := 0;
  while j < array.length do (
    if i != j then always array[i?] != array[j?]
    j := j + 1);
  i := i + 1)
```

Such a constraint could be used, for example, in the implementation of a Sudoku game [83], to express the constraint that numbers must not repeat per cell, line, or column.

The above code has two main problems, however. First, if we consider collections that can grow (or shrink), these constraints would then be incorrect — they would either have to be redacted and the loops re-executed or after adding the above constraints once any changes to the size of the collection must be prohibited. Second, many object-oriented languages include a set of application programming interfaces (APIs) to work with collections, and rather than iterating manually, a method such as `all_elements_different`, if available on the `Array` class, should work in a constraint:

```
always array.all_elements_different()
```

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

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

```
def all_elements_different()
  i := 0;
  while i < self.length do (
    j := 0;
    while j < self.length do (
      if i != j && self[i] = self[j] then return false;
      j := j + 1);
    i := i + 1);
  return true
end
```

Table 3.1.: Mapping from collection predicates to declarative representation

`every`	$\forall x \in \texttt{array}.f(x)$
`some`	$\exists x \in \texttt{array}.f(x)$
`member`	$y.\exists x \in \texttt{array}.x = y$

The above code would generate a conjunction of constraints, because the early return is `false`. The constraints in the conjunction would be for the tests to that early return, pair-wise constraining `array[i] = array[j]` to be false (with the values of `i` and `j` fixed for each constraint).

Some common predicates available on collections in Common Lisp are translated as per Table 3.1. Even though this list contains only a few predicates, in practice many languages come only with a relatively small set of collection types that are supported at a language level. Languages with a rich collection library such as Common Lisp or Squeak/Smalltalk [65] are built around a small number of types and primitive operations to access and store indexed elements in an object. Thus, implementing the special support needed to support these basic predicates enables their use in a variety of contexts, including methods that are built on top of these predicates. Examples for these are the Common Lisp methods `notevery` which simply uses a negation of the predicate used in `some` or `notany`, which is the negation for `every`.

3.3.2. Constraints on User-Defined Methods

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

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

We can of course use this method in the forward direction in a constraint, according to the rules in Section 3.2.6:

```
a := new Array(2);
a[0] := 10;
a[1] := 20;
s := 30;
always s = a.sum();
a[0] := 100
```

After the `always` constraint is executed, `s` is 30; then after the final assignment to `a[0]`, `s` becomes 120. However, the method does not work backwards — for example, we cannot constrain the sum of the array and expect the system to update one or more elements to satisfy the constraint. So the constraint in the last line below will be too hard for the system to solve:

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

We have found a design pattern for user code that works well in these situations that *can* provide something that works both forward and backward. Rather than using a method that returns the calculated sum of the elements, we eagerly update the sum as the array changes in a variable. This can be done by writing an ordinary method that sets up the appropriate network of addition constraints. The `sum` becomes an instance variable of our collection, and the implementation of that collection must take care to correctly initialize the constraint network when it is created or an element is added or removed:

```
def initialize_sum()
  if self.length = 0 then
    always self.sum = 0
  else (
    sub_array := new Array(self.length - 1);
    i := 1;
    while i < self.length do (
      sub_array[i - 1] := self[i];
      always sub_array[i - 1] = self[i];
      i := i + 1);
    always self.sum = self[0] + sub_array.sum)
end
```

The advantage for code using the collection's sum in further constraints is that it is used simply as a variable — constraints on it can work both ways, and it can even be assigned and the array changes to satisfy the constraints. We will discuss additional useful design patterns such as this in Part IV.

Summary

Our design captures design principles to control the power of the solver in object-constraint programming languages to avoid surprising or non-deterministic behavior. These restrictions force developers to declare identity constraints and value constraints separately, but in return they guarantee that object structure cannot change when solving for values. We achieve this by using two-phase solving approach where first identities and structures are determined to satisfy any identity constraints and, in a second phase, value constraints are solved without further changes to the object structures. This remove non-determinism and surprising behavior with respect to the structure of objects — structure can only change through identity constraints or assignments.

These restrictions are motivated from experience with earlier Constraint-Imperative Programming languages. However, more experience with this style of programming may uncover the need for additional rules or some rules may turn out to be too strict for certain application domains. Some applications may want to have some way to backtrack over or continue execution concurrently for multiple different solutions. In future work we plan to investigate Prolog-style backtracking to make multiple solutions accessible from the imperative paradigm. Additionally, our design supports primitive types such as strings, numbers, or booleans as well as objects with named and indexable fields. However, additional support may be required for constraints over dictionaries, streams, or other structures.

4. Key Semantic Rules for Object-Constraint Programming Implementations

This chapter presents an overview of a formal development of the Babelsberg design that can be used to guide practical implementations [38], and gives and idea of the minimal requirements on the solver and language to implement our design. The full formal development that lists all semantics rules extensively is given in Appendix A.

The semantics 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 rules in the same increments as we did the design. First, Section 4.1 presents the core semantic judgments to add constraint solving to a simple imperative language that has only primitive reals, integers, and booleans. Second, Section 4.2 discusses how to integrate method lookup and dispatch, identity constraints, and the interaction with the heap into the constraint solving processes, and briefly presents two theorems for key properties (proofs for which have been previously published [39]). Third, Section 4.3 discusses the additional requirements and limitations of our design to declare constraints over collections.

What follows is a discussion of only the most relevant technical issues, with the full set of judgments and rules given in the appendix. Besides ordinary imperative evaluation, we need to concern ourselves with a) how to model translating of constraint expressions into a form suitable for the solver and solve them and b) how to modify imperative assignment rules to re-satisfy constraints if they would become invalidated.

4.1. Primitive Types

The basic language that has only primitive types is modeled using standard imperative semantics with an environment to hold bindings from names to values and primitive control structures. 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, integers, and strings, including operations on them to do arithmetic, comparison, and boolean operations.

The first key extension over a simple imperative language is the addition of a global set of active constraints to the state (besides the standard environment). The second is the addition of a symbol (ϱ) to range over a finite and totally ordered set of constraint *priorities*, which is assumed to include a bottom element `weak` and a top element `required`. Although we used read-only annotations in the examples presented throughout Chapter 3, we do not need them to model an OCP language. (The semantics of read-only annotations are well understood and may easily be added.) As the

third addition over a standard imperative language we must add the `always` and `once` primitive statements, which declare constraints.

4.1.1. Translating and Solving

The first issue we must address to add constraint capabilities to a standard imperative semantics is to define how expressions in the language are passed to the solver and its solutions applied. In addition to a normal imperative judgment to execute through statements, we add judgments to translate and solve constraints. Translation in the imperative language can be straightforward if we define that our solver language is source compatible with the imperative language. However, to implement our "frame axioms" (cf. Section 3.1.2), we automatically generate low priority stay constraints on the current state of the world before passing any constraint to the solver. This is the purpose of the following two non-standard judgments in the formal semantics:

$$\text{stay}(x{=}v, \varrho) = C$$

$$\text{stay}(E, \varrho) = C$$

The intuition for these judgments is that, given a current state, they produce a compound constraint expression, by going over each variable binding in turn and generating a simple equality constraint between the variable and its value with the priority ϱ. Given these, we add them in a conjunction with the constraint expression and pass them directly to a solver, using a second type of non-standard judgment:

$$E \models\!\!= C$$

The intuition for this judgment is that solving a constraint `C` generates a completely new system state (`E`) that has correct bindings to satisfy the constraint. Like the syntax for solver and imperative language, we can also define the environment generated by the solver to be trivially equal to the primitive global environment. Solving itself happens in a call to the constraint solver, which we treat as a black box. Our only assumption is that the solution is optimal according to the solver's semantics, as discussed earlier.

4.1.2. Re-satisfying Constraints

The second major issue to address an implementation of OCP is that assignment may invalidate constraints, and that the solving process must consequently potentially run during assignment. Although optimizations are possible, formally we can just replace the ordinary imperative assignment rules entirely and always run the solver for an assignment. Consider the following snippet:

```
x := 2 + 5
```

A standard, imperative assignment rule simply evaluates the right-hand side, and then updates the binding in the environment to reflect that `x` is now 5. However, since `x` may have constraints on it, we replace this rule with one that also evaluates the right-hand side first, but then constructs an equality constraint `once x=5` and solves that using the translation and solving process described above.

4.2. Objects and Messages

In addition to primitive values and control structures, we also want to support constraints on objects that live on the heap, have identity, and respond to messages. The semantic rules are still mostly standard imperative rules, including rules to create new objects, access fields, send messages, or compare object identity. In addition to our rules to assert and maintain constraints, we also now add slightly non-standard structural typechecking rules to test if the constraint expressions are valid. Another important change over an ordinary object-oriented languages concerns side-effects in expressions. Since message sends are expressions, and method bodies can add constraints, both our global constraint store as well as all program state may be affected by solving operations that occur within methods. This indicates problems regarding modularization of OCP programs, which we will discuss in Section 11.2 and Section 13.1

In the previous section, we assumed our simple primitive language can, for the most part, be considered to be the same as the solver language. This is no longer possible, since there are no solvers that understand and are able to resolve heap allocations and object-oriented message sends. We do impose additional requirements on the solver, however. The solver now must be able to handle *records* and *uninterpreted symbols*, which we can use to represent objects with fields and pointer references, respectively. Otherwise the semantics still treats the solver as a black box.

The semantics for this language, although omitting collections and cooperating constraint solvers, illustrates the key principles we want for an OCP language.

4.2.1. Translating and Solving

To deal with allocations and message sends that may occur in constraint expression, we extend the set of implicit actions that occur when a constraint is asserted. The stay constraints are simply extended to also assert identity constraints for heap objects. As a second step, we add judgments to recursively translate message sends in constraint expressions into a form suitable for the solver by inlining methods and also adding constraints for assumptions about object fields and identities along the way. This way, we can eagerly unroll loops and inline methods for the solver, by requiring that relevant objects and values stay constant. This is expressed in the formal semantics using the following non-standard judgment, which represents a mix of symbolic and ordinary execution:

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

The intuition for this rule is: Given a global environment ($\mathbb{E}$), a current local scope ($\mathbb{S}$), a state of the heap ($\mathbb{H}$), and a set of ordinary ($\mathbb{C}$) and identity constraints ($\mathbb{I}$), translating expression `e` for the solver yields an updated global environment ($\mathbb{E}'$), a set of assumptions to make the inlining valid ($\texttt{e}_C$), and an inlined expression ($\texttt{e}'$). In our inlining rules, we translate local variables by generating global names for them, since we do not want to assume that the solver can use multiple scopes. Arguments to method calls are constrained to be equal to the expression that generated them, and receivers are constrained to stay identical to what they are at the moment.

As a third step before solving, we must ensure that no heap updates occur as part of executing the constraint expression, because the solver does not know about the heap. To do so, we must structurally type check the constraint expression. This process enforces our structural compatibility principles presented in Section 3.2.4 by checking whether no unbound variables occur in the expression and whether all field accesses are valid. These assertions are just checked: the system will never change anything to enforce them and if one is violated it is just an error. We require the programmer to ensure that an object with the expected fields is first assigned to a variable used in

object constraints, just as a programmer would need to ensure that an object with the expected fields was assigned to a object-valued variable in a standard language. In the formalism, these checks are represented by the following two judgments:

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

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

The intuition for the first judgment is that expression `e` has type `T` if it does typecheck given environment $\mathbb{E}$ and head $\mathbb{H}$. The intuition for the second judgment is that if it passes, the constraint `C` is well formed, that is, its constraint expression is has a boolean type. New object construction on the heap does not typecheck, since constraints must be side-effect-free. Method calls also do not typecheck: we have already inlined method invocations at this point. Finally, identity constraints do not typecheck: they are solved separately and should not appear in ordinary constraints.

Besides these two additional steps, solving essentially proceeds as before, with the solver generating a new global environment, and mapping the generated global names for local variables back into their local scopes.

4.2.2. Re-satisfying Constraints

As for the primitive semantics, the assignment rules are replaced to invoke the solver. The situation is complicated for a full OCP language, however, due to the existence of fields and a heap. For assignment, we now distinguish assignments to a new local name, an existing local name, or to a field.

For new locals, we do not invoke the solver at all, and simply use a standard rule for creating the bindings in the environment and heap. To tame the power of the solver, we add the two-phase solving described in Section 3.2.4 for the other types of assignments. For assignments to existing locals or fields, we use the same scheme as before treating assignment like a `once` constraint, but now we do it twice: the first time we solve, we omit the typechecking step. Remember that identity comparisons do not typecheck, so if we did the typechecking when assigning an object on the heap that has identity, the constraint would be rejected. The first attempt at solving thus allows updating variables that store pointers to objects. The second solving process then assumes that all identities are fixed, and we typecheck under the new environment before allowing the solution to return. This ensures that the solver still cannot generate fields or modify the structure of objects during solving.

Key Properties In prior publications, we have presented and proven two key properties of our formalism for a simpler language, called Babelsberg/UID [38, 40]. The first theorem formalizes the idea that any solution to a value constraint preserves the structures of the objects on the environment and heap. 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). Both theorems are presented in their updated form in the Appendix A.2.3.

4.3. Collections of Objects

In this final extension to Babelsberg/Objects, we support a limited number of predicates on collections. These additional rules illustrate the facilities to support higher-order functions in this

design. The semantics are a straight extension from the full object-oriented language. The main addition is the inclusion of an element Ⓟ , which ranges over the core predicates on collections such as `every`, `some`, `member`. For languages which support re-definition of the methods that come with the language, we assume that the element matches only the original implementations, not user-defined re-definitions.

This syntactic element reflects that a language implementer would have to manually check and decide which collection predicates in the language are available and safe to use in constraints. This means that the main work in implementing and OCP language with support for collections is to check the core language methods and add facilities to recognize and allow relevant predicates in constraints.

Besides this addition, supporting collections only requires some fairly minor extensions to the rules. We must extend the inlining judgment to also work for well-known collection predicates, that must be inlined over multiple statements. To support this, the constraint expressions that are returned by the inlining rule for statements are split into groups for conjunctions and disjunctions — this is required to track, based on the early returns that are encountered, whether a set of inlined expressions all need to be satisfied or if just one needs to be satisfied, as explained in Section 3.3.1.

There is a noteworthy problem with our design and formalism to determine constraints from early returns: they may generate constraints that are too strong. Consider the following method, which tests if either at least one element in the array is larger than ten, or else all elements are negative:

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

always array.some_or_none()
```

Here, the constraint would be satisfied if:

$$\exists x \in \texttt{array}.x > 10 \vee \forall x \in \texttt{array}.\neg(x < 0)$$

But our design (and formal rules) would always require the conjunction to be satisfied, so the solver would have to solve this stronger constraints instead:

$$\forall x \in \texttt{array}.\neg(x < 0)$$

This problem can be circumvented, but would make the design and the formal rules more complex. We have decided to avoid this additional complexity, because the code above could easily be rewritten to use two methods which each test one property, and then use these in a disjunction.

4.4. Executable Specifications

Although a formal semantics can elucidate many details of how a programming language design is intended to work, and proof strategies exist to show the formalism accurately reflects those details, it is as easy to make mundane mistakes in a formal semantics as it is in any formal language.

Providing an executable form of a semantics and running example programs helps catch such issues. Additionally, we would like to derive test suites from executable semantics to verify concrete implemented languages against the formal design. Regardless of whether the implementations evolved before or alongside the formal semantics or whether the implementation was created following the principles of the formalization, language specifics and optimization add complexity and make the conformance hard to show.

Creating an Executable Semantics A number of systems support practitioners in generating executable semantics from formal specifications, one of which is Relational Model Language [100]. Relational Model Language 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 Relational Model Language [35]; we created a parser for Babelsberg/Objects, implemented all the judgments and their rules as Relational Model Language *relations* and corresponding *rules*, and finally created an interface to the Z3 optimizing solver [6] to implement the $\mathbb{E};\mathbb{H} \models\!\!= \mathtt{c}$ judgment.

Translating our big-step operational semantics into Relational Model Language is straightforward. The opaque judgments for method lookup and type conversion are simply expressed as dictionary lookups by method name and Relational Model Language-level typecasts, respectively. The solver judgment, however, requires more work and requires in particular:

- Translating the syntax of the semantic's constraint expressions into Z3 syntax. This involves, besides a simple syntactic conversion, to convert operations into their type-specific equivalents in Z3, and generating declarations for the global program state, all used variables, as well as all labels and references that are used in the constraints. Furthermore, we have to encode strings and string operations (which are not natively supported in Z3).
- Defining Z3 types that encompass all types of the semantic language, to simulate a dynamically typed language with Z3's statically typed variables. This is achieved by making all variables union typed, with record, real, boolean, string, and reference type parts.
- Retrieving the solver output and generating a fresh environment and heap from the solution.

For implementation details on these steps, we refer to our technical report [35].

Generating Specification Test Suites 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 example programs given in Chapter 3 are designed to clarify corner cases and document design decisions. As part of testing our executable semantics, we have created a test suite from these programs that can automatically verify that the formal language produces the expected results. As the semantics of Babelsberg evolves, we can adapt and extend the suite.

From these programs, we also derive *executable specifications*: test suites that automatically test if the actual implementations correspond to our semantics. To that end, we have created a frame-

[1] `http://rspec.info/`, accessed February 25, 2015
[2] `http://perl6.org/specification/`, accessed February 25, 2015
[3] `http://openjdk.java.net/groups/conformance/`, accessed February 25, 2015

work to generate language-specific test suites. To use this framework, a language implementations only has to provide syntax transformation rules to convert the tests from the Babelsberg/Objects language, and a scaffold that implements the methods used in the tests. The framework provides a template for the source transformation in form of Relational Model Language rules. A language only needs to complete these rules to generate a test suite.

Summary

The key rules of OCP languages and where their semantics differ from well-known imperative and object-oriented semantics are very few. The rules we presented here implement the translation of expressions into constraints and the interaction of the imperative environment and heap with the constraint solver. The full formal development that lists these key rules embedded into a full imperative and object-oriented operational semantics is given in Appendix A.

We have found it useful to approach these key properties incrementally and develop them in conjunction with an executable version of the semantics that can run our design examples as tests. The executable version maps directly to the formal rules. Thus, any changes to the design and semantics were verified by running example programs against the executable version of the semantics as well as against our implementations, besides a formal analysis of our rules. We expect that future extensions to the semantics to formalize our design for cooperating solvers (presented in Chapter 5) can be developed in the same manner.

The semantics given here and in the appendix defines a useful subset of the Babelsberg family of object-constraint languages. Compared to our design, the semantics imposes additional restrictions in the interest of reducing the complexity of the rules. These restrictions may turn out to be too conservative for practical Babelsberg applications, and more complex rules might be warranted of they can enable more general use cases for multi-directional constraints or constraints over collections while keeping with our key properties intact.

5. An Architecture For Using Multiple Constraint Solvers

So far we have shown constraints over integers and objects referring to other objects or integer fields, but assuming the host language supports more primitive types than just integers, we need to consider whether the following should be valid:

```
x := 5;
x := ”Hello”;
```

Since we are modeling a language that should be a straightforward extension of existing object-oriented languages, including dynamic languages, we want to allow variables to be able to change their type. Thus, when we execute the above we get the following constraints when executing the second statement:

$$\begin{array}{rl} \text{weak} & x = 5 \\ \text{required} & x = \texttt{”Hello”} \end{array}$$

It is easy to see that the final result should be `x` $\mapsto$ `”Hello”`. The changed type might propagate through an `always` constraint:

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

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

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

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

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

$$\begin{array}{rl} \text{weak} & x = 5 \\ \text{weak} & y = 10 \\ \text{required} & y = x + x \\ \text{required} & x = \texttt{”Hello”} \end{array}$$

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

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

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

Here the solver is free to choose either an integer or a string for x. This may seem strange, and also not really in keeping with the spirit of the goals for our languages with respect to avoiding non-determinism. However, as mentioned in Chapter 3, we are looking to avoid non-determinism in solutions involving object identity and object structure. In this example, both integers and strings are primitive types without identity or structure, and thus the above program should be legal. However, when we consider a language where both integers and strings are boxed as objects for which identity *is* important or that have some non-identical structure (like a `length` field on a string), then the above would be illegal. Regardless, we need a solving strategy that can deal with different types or objects.

5.1. Cooperating Solvers Architecture

In practice, it is often infeasible to provide a single constraint solver that works well for all types or operations involved of a problem; instead, different solvers may be more appropriate than others for some aspects, and which need to work together to solve the problem.

Prior work on constraint-programming systems including Sketchpad [115], ThingLab [8], and Kaleidoscope [47] recognized this need for letting multiple solvers cooperate. However, these systems did not include a general architecture for letting multiple solvers cooperate, but rather included a fixed set of solvers with a meta-solving strategy that was hand-coded to connect these solvers in the appropriate way. In a similar vein, specialized solvers such as Ultraviolet [10] or DETAIL [60] combine multiple decision strategies to solve more complex problems. (In the case of Ultraviolet, these were a local propagation solver for functional constraints, a local propagation solver for constraints over reals, a solver for simultaneous linear equations, and an incomplete solver for simultaneous linear inequalities.) However, these, too were inflexible to extend.

In another avenue, there is much recent work in the area of SMT solvers [93], mostly focused on the context of model checking and program verification. As described in Section 2.3, SMT solvers communicate through shared variables, inferring equalities on those variables. This is powerful, but has certain limitations. Most commonly, a limitation is that the method is incomplete for a number of theories, but more importantly for the work presented here, it does not deal at all with soft constraints, whereas our design explicitly requires them.

Thus, we find a need to combine multiple cooperating solvers in a general way, while at the same time supporting soft-constraints and, ideally, fast incremental re-solving. Borning [7] proposed an architectural design that puts minimal requirements on the component solvers, but still allows for more powerful interactions in parts of the constraint graph. As part of this work, we have implemented and evaluated this architecture in practice.

At the top level, constraints and constraint variables are simply viewed as a bipartite graph. The constraints are partitioned into regions that are connected via read-only variables. Each constraint belongs to exactly one region, but regions may share variables. This happens if variables occur in multiple constraints that belong to different regions. Such variables, however, must then be read-only in all but one of the regions.

In this architecture, the regions must form an acyclic graph, so that solving can simply proceed from the upstream to the downstream regions, propagating variable values. This architecture prohibits loops and a system that oscillates without finding a solution. We find an order for solving the regions, so that region B is solved after region A if B is downstream from A in terms of the read-only variables. Read-only variables are represented simply through required equality constraints which are automatically generated as higher regions determine the values of variables. The result is a very loose coupling among the cooperating solvers. Furthermore, the programmer can explicitly control

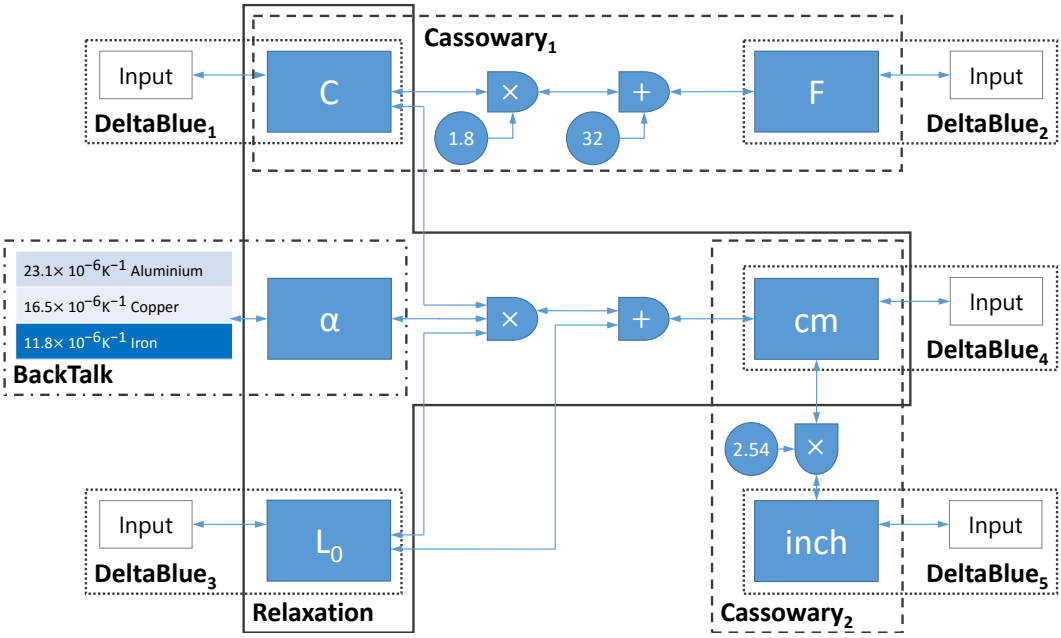


Figure 5.1.: A graph of constraints in our cooperating solvers architecture

the position of a solver in this graph, or the system can create the order without the programmer's support.

Figure 5.1 shows an example configuration. In this example, we are looking at a set of constraints for determining the thermal expansion of a metal rod. Such a system is useful to run in multiple directions, to find the expansion given certain temperature ranges, to find safe temperature ranges given maximum and minimum lengths of the metal rod, or to select the most suitable metal given some temperature and expansion. In addition, we also want to allow entering lengths and temperatures in the metric or the imperial system.

Converting between Centimeters and Inches as well as between Fahrenheit and Celsius is done by means of a linear equation, thus the constraints to do so could be handed to Cassowary. Converting textual input fields to float values and back must be done via a local propagation solver such as DeltaBlue. The calculation of the thermal expansion is a non-linear equation that multiples the temperature, the base-length at zero degrees Celsius, and the coefficient of thermal expansion specific to the selected metal. This equation is not linear, and thus cannot be solved by Cassowary, but Sketchpad's relaxation solver, for example, can find a solution. Finally, the coefficient of thermal expansion is determined by the type of metal, a finite list of associations. To solve for such a value, we need to choose a finite domain solver such as BackTalk.

User input always arrives via one or more of the input fields. Depending on which input variable(s) are provided by the user, the corresponding DeltaBlue regions are sorted first. For example, if we want to determine what length an Iron rod has at 60 degrees Fahrenheit, if its length is 100cm at zero degrees Celsius, DeltaBlue_2, DeltaBlue_3, and the BackTalk region are sorted to be solved first. They determine L_0, F, and α from the user input, consequently these variables will become read-only for all downstream regions. Since both C and cm are still undetermined now, the region for Sketchpad's relaxation has more unknowns than Cassowary_1. Thus, the latter is sorted to be

solved next. Now Relaxation and DeltaBlue_1 both have one unknown left. We proceed sorting these regions with the fewest degrees of freedom first, and end up with the following solving plan:

$$\left\langle \begin{matrix} \text{DeltaBlue}_2 \\ \text{DeltaBlue}_3 \\ \text{BackTalk} \end{matrix} \right\rangle \text{Cassowary}_1 \left\langle \begin{matrix} \text{DeltaBlue}_1 \\ \text{Relaxation} \end{matrix} \right\rangle \left\langle \begin{matrix} \text{DeltaBlue}_4 \\ \text{Cassowary}_2 \end{matrix} \right\rangle \text{DeltaBlue}_5$$

Once the solver regions are sorted, solving proceeds from the furthest upstream region. Each region will determine values for the variables it can write to, and the downstream regions will adjust to accommodate the new values propagated to their read-only variables from higher level regions. Soft constraints are solved for just within each region — in keeping with the theory of hard and soft constraints in the presence of read-only variables [11], if a soft constraint in an upstream region restricts a variable to a certain value, then a downstream region must use that value and can in fact not distinguish if this value was determined by a required or a soft constraint. If constraints in a downstream region cannot be satisfied due to an upstream soft constraint, we do not backtrack and instead fail solving entirely.

This architecture imposes severe limitations on the kinds of constraints that can be solved. For example, there is no support for solving soft constraints level by level (first solving all required constraints, then proceeding to the soft constraints), because the architecture does not assume that all solvers support solving soft constraints in this manner. However, the architecture is flexible in that a region can itself be partitioned further and use a different strategy for combining solvers, which possibly has stronger requirements on the participating solvers. In fact, it is possible to have a region that uses an SMT solver (which itself uses different solvers internally), and another region that combines DeltaBlue and Cassowary solvers and supports fast incremental re-solving within that region. For example, if we determine that a variable that changes frequently is referenced from constraints in regions that are solved by DeltaBlue as well as Cassowary, we combine this region and, when the complete constraint graph is re-solved due to an update to this variable, we automatically create edit constraints for DeltaBlue and Cassowary and run them. This way, incremental re-solving within those regions is fast, but we do not prohibit also using solvers for which no fast incremental re-solving strategy is available.

5.2. User-defined Constraint Solvers

In the Babelsberg design, all solvers use the same interface to communicate with the VM so developers can add new solvers and replace existing ones. Besides allowing users to add new solvers, it also provides encapsulation. Prior OCP languages such as Kaleidoscope treated the solving processes specially and allowed the solvers to ignore encapsulation and manipulate the contents of objects directly. Solvers in Babelsberg, on the other hand, receive the current objects as values and can return new objects, but also send messages to those objects to mutate them in place. This is particularly useful, for example, when complex objects such as files or methods that need VM support are used in constraints.

A solver can be written in the host language itself, but has to provide an interface to the runtime. A solver should declare its capabilities (to announce to the system which types of constraints it supports), but can also omit it (and must then be used explicitly, without any automatic selection). Solvers further need to provide at minimum two functions:

SOLVE A function should be available to trigger solving. This method should return all variable names known to the solver and their new values.

Add/Remove Constraint Functions must be available to add and remove constraint expression to and from a solver. These function should not to solve immediately (so that the performance heuristics can measure solving and CCM independently). Furthermore, removing constraints must be possible, even if the design does not include user-level support for disabling an `always` constraint once it has been enabled. Removing constraints is required to retract `once` constraints after solving, and to update variable values in the solver. The latter is used both for assignments and for propagating the values of read-only variables in the cooperating solvers architecture.

User-defined constraint solvers in combination with our cooperating solvers architecture are key to the kind of flexibility in constraint solving that we hope to achieve with this system, by giving the user the power to write simple, highly specialized solvers that can be combined with powerful, general solving strategies.

5.3. Automatic Solver Selection

Given that multiple solvers can be used cooperatively with this design, we need to figure out which solver to use for which constraint. In this section, we present different heuristics for selecting a solver automatically and present their trade-offs.

The solvers in ThingLab or Kaleidoscope were automatically selected based on the type of constraint that was handed to it. In these languages, however, the set of available constraint solvers was fixed. More importantly, each solver had capabilities that the others lacked, and thus many constraints could clearly only be handled by one particular solver. The algorithm which determined how the constraints were added to the solvers only had to check for these specific capabilities of the solvers, and could be optimized easily. Such an algorithm is not possible with this design, however, since we do not know in advance which capabilities the solvers have. Even worse, we may want to use different solvers with exactly the same capabilities, but that have different characteristics with regards to performance or solution quality.

The simplest approach at assigning a constraint to a particular solver would be to let the user do it. Each constraint must then be annotated with the correct solver to use in the source — if the selection is incorrect (i.e., assigning the constraint to a solver which cannot solve it) or not ideal (i.e., using a general purpose problem solver when a specialized algorithm exists), the system does not attempt to correct the problem. However, this simple approach is undesirable. OCP is supposed to provide convenient access to constraint solving for imperative programmers, thus it is inconvenient to expect each user to have to learn about the capabilities and trade-offs between the different solvers before using them. Instead, manual annotation should be the fallback if the system is unable to select the ideal solver in a situation.

5.3.1. Eager Selection by Type

For this algorithm, the solvers in the system must extend appropriate classes to respond to a solver selection message with an instance of themselves. When executing a constraint expression in CCM, the system determines the most specific dynamic types of each variable that it encounters. As long as no solver is selected for the current constraint, the system sends a solver selection message to the object to ask which constraint solver should be used for it. If the object does not understand the message, this indicates that no solver declared that it can handle this type of object. If the object responds with a solver, the constraint is immediately assigned to that solver. Any further variables

that are encountered are now interpreted with this solver. If they happen to be of a type that the solver can handle, they are normally represented as variables. If they are not, their current value will be used as if it had been annotated as read-only in the constraint expression. The solver will be able to use the value of the variable but not write to it, but when the variable changes, the updated value is passed to the solver.

If during the execution of the constraint expression no solver is selected, the constraint is taken as-is: if it is already `true`, no further action must be taken. When any of the variables that participate in the constraint change, the expression is re-executed using this same algorithm. If the expression returns `false` and no solver is selected, it is treated as unsatisfiable. In either case, a practical language should provide an appropriate warning to users, informing them that no solver could be found for a constraint involving the dynamic types that were encountered.

This selection algorithm is simple and fast. Constraints that deal with few types can quickly be assigned to the correct solver. However, the algorithm is brittle for constraints that include multiple types that each should be handled by a different solver. Consider the following example:

```
s := ”Hello”;
n := 5;
always n = s.length() * 2;
always s.length() * 2 = n;
n := 10
```

Suppose we have a local propagation solver that can deal with equalities over strings and numbers, but not arithmetic, and one that can solver linear equations over reals, but does not support strings. The two constraints in line 3 and 4 should be idempotent and it should be irrelevant in which order they are defined or solved.

The first constraint will be handed to the linear arithmetic solver, since we encounter the number `n` first during CCM. The solver will treat the string as a constant, and is thus forced to use the return value of the method `s.length()` to update `n` to `10`. The second constraint will be handed to the local propagation solver, since we encounter the string `s` first. The constraint is already satisfied, so nothing needs to be done.

When we execute the assignment on line 5, the order in which the solvers are called to solve these constraints matters. We must first call the local propagation solver to update the string length. If we call the linear arithmetic solver first, it will give up and treat the constraint as unsatisfiable, since it cannot manipulate the string to make it longer. This can lead to confusing cases where the system will say a set of constraints is unsolvable, but a simple re-ordering of operations in the constraint expressions will make it work.

5.3.2. Selection by Preference

If more than two solvers can solve the current set of constraints, it may be desirable to always give preference to one solver over another, to avoid issues where the solver surprisingly changes when a constraint expression is refactored. To assign preferences, the system maintains a list of solvers implicitly ordered by preference. This list may be generated or supplied by the user. In either case, the list of solvers should be created early in the execution of the program and, once the first constraint has been created, it should only be possible to append, not remove or re-order solvers.

Instead of executing the constraint expression just once, it is executed in CCM once for each of the available solvers. The first solver that is able to work with at least some of the variables that participate in the constraint is selected and the constraint is assigned to it. Executing the

constraint expression multiple times is safe due to our requirement on it being free of side-effects (cf. Section 3.1.4).

For the above example, suppose that the arithmetic solver is preferred over the local propagation solver. In this case, both constraints will be assigned to it. This means, however, that the assignment on line 5 now becomes unsatisfiable! While this may seem like a strong restriction on the kinds of programs that can be written in this manner, we consider it preferable over the potential surprise that the two constraints above, even though they trivially express the same equality, are assigned to different solvers. In this scenario, the constraint system will always be unsatisfiable, regardless of how we refactor the two constraint expressions. To make the program work, the user should manually select the local propagation solver in this case, or else re-order the solvers to give preference to the local propagation solver at the beginning of the execution.

5.3.3. Heuristic Selection

The preferential solver selection is heuristic: it selects the first solver that can work with at least some of the variables that participate in the constraint. This heuristic can be easily extended. An obvious extension is to select the first solver that can deal with the *most* variables, and go by preference only in case of a tie. There are more such extensions that make sense in this decision procedure. As described above, there may be solvers that have exactly the same capabilities, and that only differ in their performance or in the quality of their results.

Some of the solvers' properties, such as performance and the quality of the result, can be determined automatically. For others, this solver selection procedure requires solvers to come annotated with *capabilities*, such as the types and operations it supports, for which theories it is complete or incomplete, or whether it supports finite as well as infinite domains.

A simple example, which the preferential selection can already decide, is a constraint on two strings and the choice between a finite domain solver and an local propagation solver:

```
x := "Good Morning"
y := "G'day"
always x = y
```

We can tell from the types that the simplex solver will not be able to find a solution to this constraint and that we should pick the local propagation solver.

However, even a slightly more complex example may make our decision much more difficult. Suppose we hand the following constraint over the reals to both a simplex solver like Cassowary and a relaxation solver:

```
a := 2.0
b := 2.0
always a * a = b
```

Again reject the simplex solver, this time because it is not equipped to deal with non-linear equalities. The relaxation solver, on the other hand, approximates a solution to this constraint by linearizing it, but it would be an approximation only. More worrisome is that the solving algorithm can diverge, so even if this constraint can be solved once, it may not be possible if we change either `a` or `b` too much in one step. Suppose a relaxation solver approximates the solution $a \mapsto 2.0$ and $b \mapsto 3.9999999999998$ for the above constraint; because the relaxation method using numerical approximation, it is prone to round-off errors. If we have a solver like Z3 available, we might want to select it instead, both for higher precision, and because the solving theory is more robust (albeit not complete) for non-linear arithmetic over the reals. However, Z3 may pick another (valid)

solution $a \mapsto 1.4$ and $b \mapsto 1.96$. This solution means that the sum of the changes to the variables is smaller than with the relaxation method, but for some applications, we may prefer a solution which modifies the least number of variables instead.

Our heuristic selection attempts to weigh these different criteria. As in the selection by preference, we execute the constraint expression once for solver, and in addition we let each solver solve the constraint once, but without updating the environment and heap with the solution. Using the preference and the below heuristics in conjunction, we can then select the best solver and add the constraint to it. As constraints are added, we regularly re-evaluate the metrics for existing constraints to check if they should be moved for better performance or better results. Similarly, when a constraint becomes unsatisfiable, but the current solver is annotated as being incomplete, we also re-evaluate the metrics to search for another solver.

Precision As an 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, although the Z3 real theory works on reals of arbitrary precision, these reals will be represented in many languages as finite-precision floating point numbers, so again round-off errors may occur. The precision of the results these solvers produce may thus differ in practice, and could be considered as part of the solver selection.

Variable Changes Another property that may impact the quality of the solution is how the solver implementation affects the selection of solutions when multiple solutions are available. Although stay constraints ensure that any solution is close to the previous state of the system, there are often multiple possible solutions that satisfy this property. Which solution is selected is usually an artifact of the solver implementation. However, problems such as the split-stay problem may make some solutions still more desirable than others, and solvers which avoid it may be preferable over those that do not. Similarly, solvers that change multiple variables, but achieve a small squared distance from the previous state of the system may be more or less suited to a particular problem than those that change few variables, but those by a larger amount.

Dimensionality Besides the types of variables, the operations on them and how they are connected also play a role. Cassowary can only solve linear equations, while a relaxation solver can find solutions for higher dimensional arithmetic. Thus, the operations and associated operands within a constraint expression must be considered to select the solver.

Completeness Another issue arises with incomplete theories. SMT solvers, for example, require the different partial solvers within to be convex for the decision procedure to be complete. For example, the theory $(\mathbb{Z}, +, =)$ (addition and equality over the whole numbers) is convex, but $(\mathbb{Z}, +, \leq)$ is not — a constraint involving inequalities over the integers may be satisfiable for a specific set of values, but another set of values may make it simply too hard for the solver. In that event, another solver (e.g., using a form of relaxation) may be able to approximate a solution and the constraint should be moved to that solver.

Performance Depending on the implementation, solvers may differ significantly in both base performance and complexity in required memory as well as time. As constraints are added during the execution of a program, a solver that is fast for few constraints may, due to higher complexity, become slower than a constraint with worse base performance but better complexity. A selection

algorithm can take into account how fast a solver was able to satisfy a constraint, and continues to monitor solvers to notice drops in performance when constraints are added.

In practice we have found a mixture of manual and eager selection to produce good results once the user has a basic understanding of the available solvers and is aware of the eager selection process (in particular, how seemingly idempotent refactorings may lead to different results). We have found that while preferential selection avoids some surprising behavior, this behavior occurs rarely in practice. The heuristic solver selection procedure, although very powerful, has strong drawbacks: its complexity impacts the performance of creating new constraints, and requires constant re-checking. As with any heuristic, the results also depend on the weight that is given to each of the metrics and may need to be adjusted depending on the application domain. An educational advantage, however, is that practical implementations can use the metrics to communicate to the user the selection process and increase the user's understanding of the capabilities and limits of available constraint solvers. As a guideline to implementers, we suggest using the heuristic procedure as an educational tool and guiding instrument at development-time, but revert to manual and eager selection when many constraints are added dynamically throughout the run-time of the program, to avoid the performance overhead. We present implementations of all selection procedures in Part III.

Summary

Cooperating solvers make constraint programming useful for a wider range of problems. Our design is simpler than other schemes for cooperating solvers and supports black box solvers with very few requirements on the solvers themselves. Any solver that supports soft-constraint and making variables read-only can be used with this design. This comes at the cost of flexibility in the solving process, because solvers communicate only via absolute variable values and cannot, for example, propagate ranges of valid variable values across regions. For performance reasons we also do not backtrack over solver regions and instead fail solving entirely. We plan to revisit this decision in future work to see if backtracking may be viable for some problem domains.

An advantage of our architecture is that it makes it easy to add solvers for special domains and have them work in conjunction with more general ones. This makes it possible to have a large number of solvers in a system, some of which may be able to address similar problems. We presented mechanisms for automatically deciding which solver to use for a particular constraint. How these mechanisms can be integrated with development tools is also an area of future work.

Part III.

Implementing Object-Constraint Programming

6. Declaring Constraints in an Object-oriented Language

In this part, we present four issues that arise when implementing the Babelsberg flavor of Object-Constraint Programming (OCP). This chapter discusses the issue of how to implement the constraint construction mode that translates expressions in the host language into constraints that can be passed to the solver. In Chapter 7 we elaborate on the issue of how to solve these constraints and translate the solution back into the program, how to implement our design for cooperating solvers, and how assignments trigger constraint solving. Finally, in Chapter 8 we address implementation issues related to the performance of practical OCP languages and present benchmark results.

To evaluate our design using different implementation strategies, we have created three concrete instances of it. The first, called Babelsberg/R [33] is based on the Topaz Ruby VM [31, 112, 41]. It provides full integration with all types of variables in Ruby and integrates with a modern JIT. In the next chapters, we will use this implementation to illustrate those issues common to all implementations.

Where appropriate, we compare the VM-based implementation approach to our two library-based implementation in pure user-code. One drawback of a VM-based approach is that, although based on a particular host-language, the OCP extensions will only work on the customized VM. However, in many languages that are of interest to the industry and research communities, applications typically have to work on a variety of client VMs. Thus, using a library-based approach, we created Babelsberg/JS [34], implemented on top of JavaScript in the Lively Kernel [66] environment, and and Babelsberg/S [52], based on Squeak/Smalltalk [65]. These implementations run on all modern VMs for their respective host languages.

In this section, we first provide an overview of the key concepts of the implementations before plunging into the details.

Babelsberg/R A full implementation of object-constraint programming requires deep integration with the underlying language to support all features of the host language, including different kinds of variables and full access to closure scopes to correctly construct constraints from constraint expressions. One way of ensuring this kind of access is a VM-based implementation such as Babelsberg/R.

The basis of Babelsberg/R is Topaz Ruby, an experimental VM built using the *PyPy/RPython* toolchain [104], a restricted, object-oriented subset of Python. Topaz is a Ruby bytecode interpreter. It is written in an object-oriented fashion, with each bytecode implemented as a method of the VM-level 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. The RPython translation annotates the interpreter to combine it with a fast, generic JIT compiler and garbage collector. In Babelsberg/R, we leverage the object-oriented design of Topaz to implement OCP and inherit the JIT and garbage collector from RPython. The semantic model is thus a pure extension of Ruby's,

and supports all of the existing Ruby constructs such as classes, instances, methods, message sends, and blocks (closures).

Babelsberg/R requires two semantic changes to support constraint creation and solving. First, we extend the Ruby standard library with the `always` primitive that receives a closure, and creates a constraint from it. The constraint is, that the last expression in the closure should return true. The closure can reference objects, invoke methods on them, express assertions about their identity, type, or protocol, and use assignments and control structures. The syntax to invoke this primitive is the same as for an ordinary global method that receives a block closure argument. The restrictions on what kinds of expressions can be used in the closure are the same as in the formal design (cf. Section 3.2.6), and are dynamically checked during CCM.

As the second extension we add subclasses of the Topaz `Interpreter` that implement the different interpreter modes for constraint construction, imperative evaluation, and constraint solving. 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 by using the VM internal representations of objects and scopes to access variables directly while ignoring encapsulation in the VM. 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. In imperative evaluation mode the interpreter operates in the standard fashion, except that LOAD and STORE operations on 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). Finally, in constraint solving mode, the interpreter operates in the fashion of an unmodified Ruby VM, that is, it does not intercept variable accesses to trigger constraint solving. Since our design for cooperating solvers enables a wider range of applications by combining specialized solvers, a goal of a practical implementation is to make it easy to add multiple constraint solvers as libraries to the host language. When these are used for solving constraints, however, the inner workings of the solvers should not recursively trigger constraint solving. We describe an architecture that makes it straightforward to add new solvers and which does not privilege the solvers provided with the basic implementation (they are simply the solvers that are in the initial library).

Babelsberg/JS and Babelsberg/S The Babelsberg/JS library is developed in the Lively Kernel, but is itself written in pure JavaScript and can be used independently of the Lively Kernel environment, on any modern JavaScript VM. Babelsberg/S is based on Squeak/Smalltalk. The two implementations are sufficiently similar that we explain only the first. Babelsberg/JS has a number of issues that Babelsberg/S does not have, due to the fact that, compared to JavaScript, the Squeak language is a more flexible basis to implement language extensions.

The Babelsberg/JS library provides three main parts: Like the VM-based approach, Babelsberg/JS approach uses different modes of execution to interpret and solve constraints, so the first part is a JavaScript interpreter written in JavaScript is used to evaluate constraint expressions and implement the constraint construction mode. This is required because popular JavaScript VMs[1] (Apple Safari's SquirrelFish[2], Google Chrome's V8[3], Mozilla Firefox's SpiderMonkey [51], or Microsoft's Chakra[4]) do not provide direct access to the native interpreter or execution context of the caller, so we cannot re-use or adapt parts of the underlying interpreter as in Babelsberg/R. The only thing close to what

[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)

we need in JavaScript is `eval`, but we cannot use that if we want to support calling user defined functions in constraints.

Second, Babelsberg/JS includes pure JavaScript implementations of all types of constraint solvers presented in Section 2.3. While in the VM-based approach we offer an interface for solvers not written in Ruby to be used with Babelsberg/R, we cannot do so in a JavaScript library.

Third, we provide tools to make developing Babelsberg applications in the Lively Kernel environment easier. While the VM-based approach customizes the syntax and adds a primitive to declare constraints, we cannot do so in JavaScript. However, we do provide a source-to-source transformation so constraints can be written in a more convenient syntax (that is also valid JavaScript), but when we interpret the code as Babelsberg code, we transform it. Note that while this transformation makes writing constraints and adding them to *parts* [77] in Lively Kernel more convenient, it can still be used outside the Lively environment, and can be used, for example during a pre-compilation step.

6.1. Implementing Constraint Construction Mode

All our implementation approaches implement CCM with a custom `ConstraintInterpreter`, either as part of the customized VM or as a interpreter written in the host language.

Babelsberg/R extends the VM with new primitive methods `always` and `once`. These take an ordinary Ruby block closure argument, which is used as the constraint. Besides those new methods, we only include a minor syntactic extension to conveniently mark variables as read-only in constraints. This is expressed by sending the question-mark method (*?*) to a value in a constraint expression[5]. As in the design, a read-only variable can only be assigned to, but cannot be changed by the solver to satisfy a constraint it appears in. This is useful, for example, for parameterized constraints so the solver knows not to change the parameter:

```
class Rectangle
  def fix_size(area)
    always { self.area == area.? }
  end
end
```

Without the annotation, a solver may choose to change the local variable `area` rather than the possibly the the `x` and `y` values of one of the corners of the receiver, if, for example, the solver tries to minimize the number of variables that are adjusted to satisfy a constraint (as is often desired).

Babelsberg/JS provides two ordinary methods, `always` and `once`, which receive a JavaScript closure and an object that captures the local scope as an argument. Since we cannot extend the syntax of the host language, read-only annotations are instead implemented as a global function `ro`. Below is the same example of a parameterized constraint in Babelsberg/JS:

```
function(rectangle, options) {
   always(function() {
            return rectangle.area == ro(options.area)
          },
          {scope: {rectangle: rectangle, options: options}});
}
```

An important difference to the VM implementation is that we need a second argument that explicitly captures the local scope. Since JavaScript does not provide reflective access to a functions

[5] Outside of a constraint expression, this method raises an exception, since it does not have sensible semantics.

defining scope, our custom interpreter cannot look up names used in the constraint expression in the caller's environment. Instead, the names and values in the scope have to be passed explicitly. [6] For convenience Babelsberg/JS provides a source-to-source transformation based on *UglifyJS*[7], 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, and could also be integrated it into the deployment tools used in other JavaScript Integrated Development Environments (IDEs). With the source transformation, we can potentially introduce any DSL we want, however, we have chosen to use a syntax that is still valid JavaScript. Using the transformation, the above example becomes:

```
function(rectangle, options) {
   always: { rectangle.area == ro(options.area) }
}
```

Requiring either a source transformation or explicit enumeration of names and values in the scope is not an inherent drawback of a library-based Babelsberg implementation, however. In Babelsberg/S, we can use the Squeak/Smalltalk reflection protocol to get programmatic access to the scope and need neither the extra argument nor a source transformation.

In both cases, the constraint expression is passed to the `ConstraintInterpreter`. The interpreter works almost like an ordinary host language interpretation, with a few changes.

6.2. Solver Selection

Before we can translate an expression in the host language into a representation suitable for a solver, we need to know which solver to translate for. In Section 5.3, we discussed multiple strategies (besides manually specifying which solver to use).

Babelsberg/R implements explicit solver selection as well as eager selection by type. If no solver was explicitly selected, the VM sends the `for_constraint` message to each variable value encountered during constraint construction. 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 `suggestValue`. For example, the Cassowary solver extends the Float class:

```
def for_constraint(name)
  Cassowary::Variable.new(name: name, value: self).tap { |v|
    Cassowary::SimplexSolver.instance.add_stay(v) }
end
```

This method creates a new variable, adds implicit, low-priority stay constraint (cf. Section 3.1.3), and returns the solver-specific variable object. The VM's solver object then sends messages to this object instead of the Float object in the context of the constraint execution.

[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.

[7] http://lisperator.net/uglifyjs/, accessed November 11, 2015

Babelsberg/JS implements some of the heuristics for solver selection we described in Section 5.3.3. If no explicit solver was selected, the constraint is handed to multiple solvers in parallel. In Babelsberg/JS, each solver can declare capabilities, such as which types and operations it supports. Based on an initial analysis of the types and operations encountered in the dynamic extent of the constraint expression, solvers a filtered, and the remaining solvers all run to solve the constraint. The solvers to finish without error are compared first for accuracy of the result and then performance to select the final solver. Currently, no further heuristics are implemented, but the order of the trade-off (performance versus accuracy) can be swapped by the user.

6.3. Accessing Variables in Constraints

Whenever a variable is accessed in a constraint for the first time, it is automatically converted into a so-called *constrained variable*, i.e., variables that can be solved for in constraints. 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. In JavaScript, we have locals, globals, and fields; of these we only allow constraints on fields. Host language variables which we do not consider as constrained variables are used as constants in the constraint, and cannot be solved for.

Converting an ordinary variable into a constrained variable is done by attaching a *solver object*. In the Ruby VM, this means that the same name can refer to two objects in the VM, one an ordinary Ruby 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.) These solver objects are VM objects that keep track of solver-specific representations of the current variable value.

Internally in Babelsberg/R, the solver object is attached to the VM structure for the type of variable. Locals are stored as *Cells* in the VM — a group of Cells being the representation of a scope. In Babelsberg/R, these cells have a field to store a *solver object* for the variable. Instance and class variables in Topaz are stored in maps [16]. For Babelsberg/R, we have extended these maps to the store solver objects as well as ordinary objects. Both storage extensions are implemented in a way that once the JIT has warmed up, read access to solver objects is no slower than access to ordinary objects (cf. Chapter 8).

In the library-based implementations, only fields have are converted. For each field accessed during the execution of a constraint expression, property accessors are created that wrap the fields. For languages like Squeak that do not have property accessors, we wrap the accessor methods instead. The wrappers delegate any access to a *ConstrainedVariable* object that manages the communication with the various solvers and create solver specific representations of the field value.

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.

6.4. Operations on Variables

When a variable is read, rather than returning its value, its associated *solver object* is returned. These objects respond to the same messages as the ordinary variable value, but instead of just executing

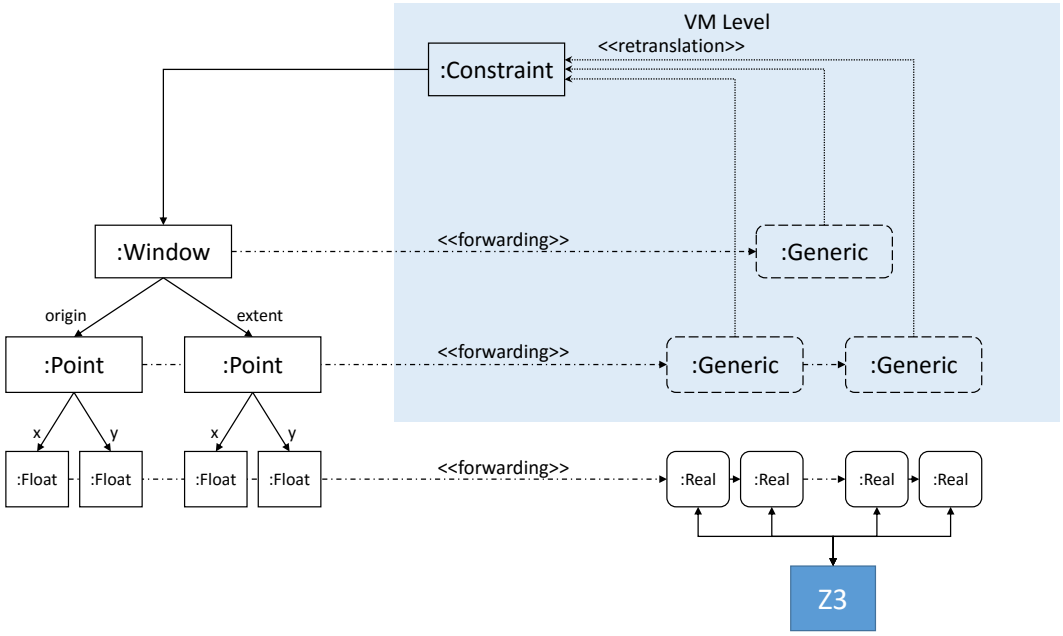


Figure 6.1.: Objects are connected through instance variables. When a constraint is constructed, their parts become connected to solver objects. A variable that is connected to a solver object delegates all accesses to that object.

the original code, they construct the relations between variables that make up the constraints as per the inlining rules in Table A.4.

To that end, during construction, the solver selected to handle the current constraint is asked to provide a representation of the variable by sending the message `constraintVariableFor` with the value as argument. As operations are performed, the solver object delegates to the active solver to create solver-specific representations of the relation. For example, using Cassowary, the expression `a <= b` would not simply return a boolean. Instead we create a `CassowaryVariable` each for `a` and `b`. These respond to the `==` method and return a `CassowaryLinearEquation`.

When the currently active solver cannot provide a specific representation of a variable, rather than adding a *specific solver object* and delegating operations to it to build relations, we add a *generic solver object*. This is only a proxy wrapper and actually executes the operations on the underlying value, effectively making that variable constant for the solver. The generic solver object is used as a marker to update that constant when the variable later changes. Generic solver objects are known to the runtime and are kept at their current value by the runtime. This allows us to use constraint solvers that do not understand uninterpreted symbols and records, since the runtime itself will ensure proper assignment and updating semantics for records and record fields, as required our design (cf. Section 3.2.5).

Primitive Objects and Operations

In Ruby the booleans, integers, or reals are proper objects, in this mode, they are treated as primitive values, since identity is not important for them. That also means that operations on them that in Ruby are implemented as methods are here simply translated to the solver language, rather than executed. In contrast, JavaScript does have primitive types and operations for arithmetic, equality, inequalities, and conjunction. During CCM, if an operand is a constrained variable or an expression

involving constrained variables, these operations are transformed into message sends so they build the relations as in Ruby.

Operations on Solvers in the Host Language

To support solvers written in the host language, the `ConstraintInterpreter` needs a way to distinguish code that should be executed in constraint construction mode from code that should not. To support this, solvers written in the host language should be subclasses of a marker class `ConstraintObject`.

During constraint construction, code in the dynamic extent of a method on those objects is evaluated in the ordinary host language interpreter. This prevents sends of `suggestValue` from causing recursive calls to the solver, and is necessary to support solvers written in the host language itself. However, this also implies that solvers themselves cannot use constraints in their implementation. This mechanism allows solvers provided by Babelsberg implementations to simply be libraries that extend core classes.

Simulating Value Classes

Another issue is that the languages we have chosen as host languages do not include value classes. Thus, we include some support for using and creating ordinary objects in constraints, as described in Section 3.2.3. Assigning to variables in ordinary class constructors used in constraints is allowed and creates additional equality constraints that are added to the constraint system. This allows us 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 in constructors encountered during constraint construction must be in single assignment form, because otherwise we will very likely create multiple contradicting equality constraints.

6.5. Constraint Construction Example

To illustrate how our implementation supports constraints on objects, recall the Window example from Section 2.2 and consider the following constraint:

```
always { window.area >= 100 }
```

This constraint asserts that the area of a Window should always be greater than or equal to 100. The assertion is expressed by sending the `area` method, 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 `window` variable is replaced with a generic solver object, since it is unlikely that we have a solver that supports such objects natively. This generic solver object asserts that the window variable cannot change and delegates the message `area` to the window object, as per our semantic rules for calling methods. The `area` simply returns the expression `extent.x * extent.y`, so we use it to create a multi-way constraint. The `extent` is also replaced with a generic solver object, and the x and y values (floats) are replaced with solver objects that have a solver-specific representation associated with them (for example Z3 variables, cf. Figure 6.1). The message `*` to the float values return symbolic expressions rather than calculating the current area of the rectangle. The expression representing the area of the rectangle is then sent the message `>=` with 100 as its argument and returns an inequality expression.

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, as well as the constructed expressions. The specific solver objects and expressions are interpreted by the solver, the generic solver objects are passed as constants. The solver then updates specific solver objects directly to satisfy the constraints. If it cannot, it raises an exception, otherwise the new value becomes the value of the variable.

Summary

Our prototype implementations each use a customized interpreter for the host language to partially execute and translate constraint expressions into a form suitable for the solver. This design works well for us, but comes with significant overhead and thus the initial creation of constraints can be slow. In future work, we will investigate mechanisms such as context- or aspect-oriented programming to replace the custom interpretation of constraint expressions. One issue to address here is the question of control structures that in most languages do not correspond to message sends that can be easily layered. Even in Smalltalk, the `ifTrue:ifFalse:` and `whileTrue:` messages are compiled to a jump bytecodes and thus cannot be easily layered.

A second potential issue of our implementation is that it relies on the existence of a parallel hierarchy of constraint object types that corresponds to the host languages primitive types. The classes in the parallel hierarchy respond to a subset of the corresponding host language type's interface. This ensures that, during constraint construction, these objects can be used interchangeably for valid constraints. While this makes it easy to add user defined constraint types, this correspondence makes it cumbersome to express additional properties that are not supported on the host language types. For example, while most object-oriented languages hide the precision of numeric types, many solvers explicitly allow the user to configure the desired precision of solutions.

7. Solving and Maintaining Constraints

Once we have the ability to declare constraints and translate them to the representation of a solver, we need to solve them. Solving happens both after a constraint is declared using either `once` or `always`, as well as after assignment. Since the process of solving is the same in both cases, we only describe what happens during assignment.

During normal execution, we intercept reads and writes for constrained variables, otherwise the host language semantics is unchanged. Whenever a variable is read, the LOAD instruction in Babelsberg/R or the wrapper in the library-based implementations must check if we are dealing with a constrained variable, and if so, read its solver object and return the value assigned by the solver. For specific solver objects, the `value` method extracts the value the variable should have from the solver's internal representation and, on the first read, copies it to the ordinary storage for that variable for faster access. For generic solver objects, the `value` method just returns the underlying variable value. As long as all constraints are satisfied, this difference is not visible to the programmer.

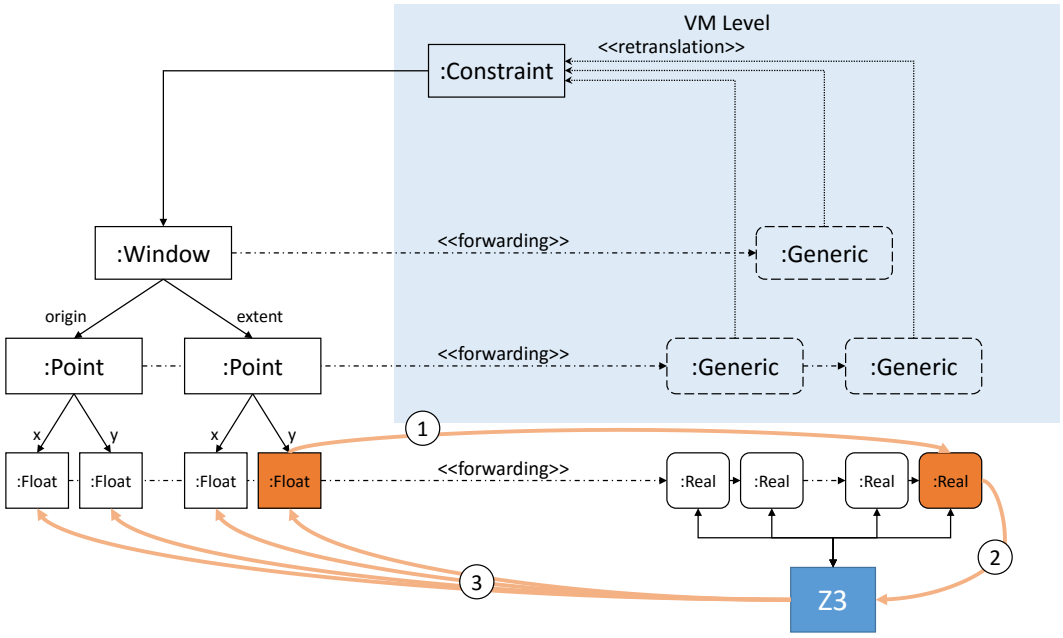
7.1. Assignment

Whenever a constrained variable is assigned to, we need to re-satisfy the constraints. In the VM-based implementation, we can intercept any assignment and check if the left-hand-side is a constrained variable, but an important limitation when implementing Babelsberg as a library seems to be that local variables cannot be solved for in constraints. In both JavaScript and Squeak, we cannot intercept variable access to local variables without rewriting the calling method, and we are not aware of commonly used languages in which such interception of local variable writes would be possible. Thus, only fields can be constrained in library-based implementations of Babelsberg.

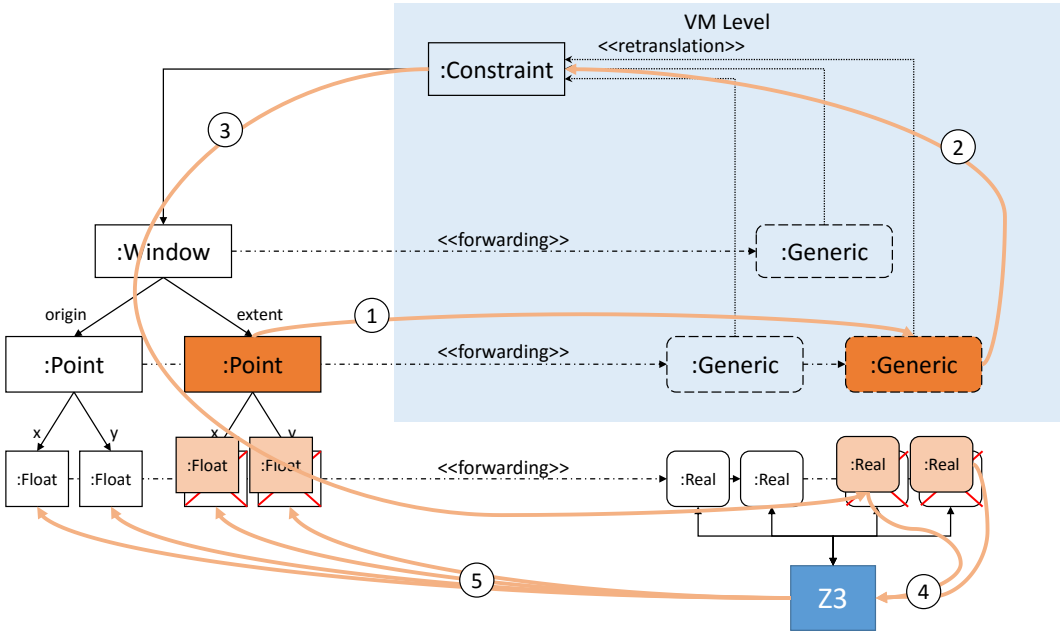
For assignments to variables that occurred in constraints, instead of changing the value of the variable, the solver object for the variable receives the messages `suggestValue`. In contrast to direct, destructive assignment in a normal imperative programs that always succeeds, `suggestValue` may raise an exception. This exception is propagated by the VM, and should be handled by the programmer. If an assignment fails in this way, the variable retains its original value.

To support the architecture for cooperating constraint solvers, 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. When a variable is assigned, we thus gather the solvers for the variable and all variables it is connected to and sort them into regions. Note that we cannot optimize the solving process to only call those solvers that know the variable, but we have to look at the transitive closure of all variables connected to the assignee. Consider the following (contrived) example:

```
a = 10
b = 10
always { a + b == 20 }
always { b * b == 100 }
a = 5
```



(a) Assigning an object that has a specific solver object forwards the assignment ①, asks the solver to re-satisfy the constraints ②, and then propagates any new values ③.



(b) Assigning a generic object ① requires retranslating the constraint ②, creating new solver objects and collecting old ones along the way ③. Afterwards, solving ④ and propagating new values ⑤ proceeds as before.

Figure 7.1.: Two cases for assigning to constrained variables.

Suppose that the first constraint is solved using Cassowary, and the second using Z3. When `a` changes, Cassowary is called to re-satisfy the first constraint, which succeeds. However, we also need trigger Z3 to (attempt to) solve the second constraint. If we do not, execution would continue in an inconsistent state with the second constraint unsatisfied, or Z3 might just change the values of `a` and `b` back to what they were on the next statement.

Assignment proceeds by first calling the defining solver. We distinguish two cases:

a) Assignment to specific solver objects, i.e., variables with values for which a solver is available, triggers the defining solver to re-satisfy the constraints. If it fails to satisfy its constraints using the new value, an exception is raised. This case is shown in Figure 7.1a, where a write to a float simply causes Z3 to check the update against the corresponding real representation in the solver.
b) Assignment to generic solver objects, i.e., variables with values for which no solver is available, invalidates all constraints these objects participate in. Since their values were treated as constants when creating constraints involving them, these constraints need to be re-evaluated. The invalidation retracts all constraints and re-executes their expression closures to create new constraints. This means that the constraint expression may be re-executed multiple times during the runtime of the program. (This is one reason why any side-effects in constraints are disallowed in the design, and should at least be benign in this practical implementation.) This case is shown in Figure 7.1b, where replacing a `Point` requires re-translating the constraint to construct new specific solver objects the float fields of the new `Point` object.

For variables with only one or no solvers that handle its type, we end here. However, if more than one solver relates to the assigned variable, we proceed by walking the acyclic, directed graph of solvers starting. For each solver, we mark any upstream variables as read-only by requiring their values to stay the same. If any of the downstream solvers fail to satisfy their constraints with the new value, an exception is generated and the assignment fails.

7.2. Changing the Type of Variables

Most solvers only provide support for a small 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.

This makes changing the type of a constrained variable potentially slow. In our prototype implementations the `ConstraintInterpreter` is not optimized for performance (because it is run only at constraint construction) — recalculating many constraints is much slower than running the same code natively. If any of the constraints fail, we have to re-assign the old value and recalculate all constraints on the variable once more.

We believe this performance penalty to be acceptable for two reasons: first, it is likely to be infrequent, as even in dynamic languages most variables are only assigned once [16]. Second, optimizing just-in-time compilers in current VMs usually assume the same, and deoptimize if this assumption is violated. For good performance in a dynamically typed program, it is thus not desirable to have variables change types often, regardless of whether they are constrained or not.

7.3. Assigning Multiple Variables

The default behavior is to invoke constraint satisfaction immediately whenever there is a change to a variable, as per the semantics. However, an important issue identified in previous OCP languages is that in practical applications, we sometimes want to update to variables at the same time, and only then invoke solving. One way to do this is to use a `once` constraint, that constrains both variables to their new value. Kaleidoscope allowed the programmer to write blocks of code during which constraints could be temporarily violated, to update multiple variables and only afterwards invoke the solver. This can be problematic, if the constraints cannot afterwards be satisfied — depending on the complexity of the code in the block, it is not always clear if the system can be returned to a consistent state.

To allow a sequence of assignments to be made with solving invoked only after all assignments are made, but still guarantee that the system can be returned to a consistent state if the constraints cannot be satisfied, we allow multi-assignments of simple values.

Babelsberg/R uses Ruby's multi-assignments to store values into multiple variables in a single statement. If multiple variables that have constraints on them are assigned in this fashion, all new values are used at once when the solver is triggered. If any constraints cannot be satisfied, all assignments fail. This way, if the exception is caught, the program is not in violation of any constraints.

In Babelsberg/JS and Babelsberg/S, we have to revert to a more expensive implementation, because these languages do not provide multi-assignment as Ruby does. To assign to multiple variables, we created a special method which takes as arguments the objects, names of their fields, and new values for these fields, and then explicitly creates a set of equality constraints that are handed to the solver at once.

7.4. Encapsulation and Solving for Objects

In the case of objects for which identity is not important (such as floats and integers), we have seen that the VM can directly return the value as determined by the solver. However, for mutable objects, identity is important and as per the semantics, the solver should only be able to update the primitive leaves of objects when solving value constraints. Furthermore, we do not want to allow the solver to disregard object-encapsulation when changing parts of objects. In some cases, such as VM objects that only expose an OO API (DOM objects, file handles, ...), it is not even possible for a solver written in the host language to disregard encapsulation — instead, it must call the appropriate API methods to mutate the object.

Default Propagation Methods Solver libraries can provide a hook on their solver objects, called `assign_constraint_value`, that will be used to update the original object with new content, this way keeping object identity intact. This provides a default local propagation method to let solutions flow into objects. For example, 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 solver object of the type `NumericArrayConstraintVariable` exactly 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
```

The `NumericArrayConstraintVariable` 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. These are implemented to generate constraints as defined in our semantics for constraints on collections of objects (cf. Section 4.3).

Below is an example that uses this to assert constraints on TWP-encoded[1] 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} }
```

Heuristic Propagation When no default propagation method is given, we use a heuristic that attempts to discover and use the appropriate setter methods. For example, consider the following constraint that the 2D point positions of objects `a` and `b` should be equal (where the equals method on points returns a conjunction of the equality checks for the `x` and `y` parts):

```
always: { a.getPosition().equals(b.getPosition()) }
```

Our heuristic is based on the idiom for getters and setters to start with `get` and `set`, respectively. In this case, we check if a method `setPosition` exists. Suppose it does, and the solver wants to change the `x` and `y` parts of `a`'s position. Rather than updating the object directly, it will create a new object of the same type (a point) with the correct values, and set that using `a.setPosition(newPoint)`.

Summary

We solve all constraints whenever a new constraint is added or a constrained variable is re-assigned. To intercept the latter, our current prototypes rely either on access to the VM or property wrappers in the language. An alternative implementation strategy using proxy objects was not explored due to the perceived overhead of full proxies versus wrapping just the necessary fields.

Simply intercepting assignments to trigger solving is enough to satisfy our design, but the practical implementations highlight other issues. The design translates all constraints for the solver every time. In the practical implementation, we can avoid the performance overhead of this process when the structures and types of variables used in constraints have not changed. Thus, we only have to do the full translation in the comparatively rare case where the type of a variable or the definition of a method used in a constraint changes.

Another issue we address only in the implementations is the question of encapsulation. Our design and semantics assumes that the solving process can ignore encapsulation and update object

[1] `http://www.dcl.hpi.uni-potsdam.de/teaching/mds/twp3.txt`, accessed November 11, 2015

fields directly. In practice, external data or primitive behavior often necessitate the use of accessor methods. The implementations allow solvers to define default propagation methods in such cases, which will be called with the new values. We also use a simple, language-specific heuristic to find accessors and we will investigate how to improve it in future work.

8. Performance of Practical Object-Constraint Programming Languages

A goal of object-constraint programming with Babelsberg was that it should be a language extensions, but with full backwards compatibility to plain object-oriented code. As a consequence, it is desirable that users of Babelsberg/R, for example, do not pay a performance penalty for the constraint extension in code that uses only unconstrained Ruby objects. To achieve this in Babelsberg/R, we interface with the RPython meta-tracing JIT compiler framework to remove the runtime overhead of the language extension for code that does not use constraints. For the library-based implementations, we rely on more traditional optimization strategies such as caching. Furthermore, the performance when constraints are used may be lower than a purely imperative solution, but it should not be so slow as to outweigh any benefits in expressiveness and maintainability. To that end, incremental resolving, a solver feature that improves constraint solving, is available in our practical implementations.

In this chapter, we will discuss these issues and present benchmark results. The full code for the benchmarks is given in Appendix B. The benchmarks were run on an otherwise idle Ubuntu 14.10 system using an Intel i5-2520M CPU forced to a constant clock speed of 2.5 GHz with 16 GB of RAM. Benchmarks were run a varying amount of iterations (given in the charts) depending on the benchmark, and each benchmark run was repeated ten times, with the mean and standard deviation shown in the results.

8.1. The Execution Overhead of Constraints

Our different implementation strategies for Babelsberg and their overhead are largely independent from the choice of solvers. The performance of constraint solvers varies wildly, and the choice of solvers has a significant impact on the performance of a given Babelsberg, but the goal remains to minimize the overhead of the framework itself.

The first step to minimize overhead is the distinction between constrained variables and ordinary variables as described in Section 6.3. This way, the interpreter first checks if a variable is dynamically related to a constraint and if it is not, it can directly return the value stored in memory rather than having to ask the solver for the current value. However, this check is performed at every variable access, which still represents a performance issue.

Second, although semantically all Babelsberg constraints are handed to the solver on each assignment, prior work on Kaleidoscope and Turtle has shown that this is detrimental to performance. One way our practical implementations of Babelsberg are able to achieve performance close to the unmodified language runtime if no constraints are relevant in a given piece of code is by omitting the solving step if no constraints dynamically relate variables in a particular piece of code and that code is optimized by the JIT. In that case, the code can be executed without calling out to the solver, eliminating any overhead. In addition, as described in Section 7.1, we only need to trigger solving for those constraints that relate variables in the transitive closure of all variables connected to the assigned variable. This further reduces the required overhead.

Integrating with the JIT

Babelsberg/R is written in RPython, which includes debugging support to print the generated JIT code for loops in the interpreter. RPython JIT-code is a list of function calls without loops and in single-static assignment form. Each JIT function generates some assembler code, and the most common ones check conditions that must be true for the following assembler code to be valid, access and manipulate memory, or to do basic operations of primitive values like integers, floats, strings, and booleans. In basic Topaz, reading a variable is translated into the following JIT code:

```
guard_nonnull_class(p1, 12345, descr=<Guard0xabcdef123456>)
p2 = getfield_gc(p1, descr=<FieldP Cell.inst_w_value>)
```

The first line represents a check that pointer `p1` is not null and of type `Cell`. If that check fails, the JIT code following it cannot be used and the VM reverts to pure interpretation. The second line represents the generated assembler to access the `w_value` field from the object that `p1` points to.

In Babelsberg/R, converting an ordinary local variable to a constrained variable was initially implemented by adding a field to store solver objects to the VM-level `Cell` class. Consequently, each reference to a variable had to check if the variable has been used in a constraint by testing if the new field is not null. The JIT compiler generated the following intermediate code:

```
guard_nonnull_class(p1, 12345, descr=<Guard0xabcdef123456>)
p2 = getfield_gc(p1, descr=<FieldP Cell.inst_w_constraint>)
guard_value(p2, ConstPtr(null), descr=<Guard0xabcdef234567>)
p3 = getfield_gc(p1, descr=<FieldP Cell.inst_w_value>)
```

Line 1 again checks that the pointer variable `p1` is not null and of the right VM-level class. Line 2 reads the `w_constraint` field of the object pointed to by `p1`. This variable holds a reference to a constrained variable object if the variable has been used in a constraint, otherwise it is `null`. For the case where the variable has not been used in a constraint, line 3 checks that this field is `null`, and only then is the variable value read.

As an optimization, instead of setting a field on a `Cell`, we introduced a new subclass of `Cell` for variables that have been used in a constraint. This unifies the class check with the constraint check, and the resulting JIT code requires one fewer guard, generating no more code than the basic Topaz VM, checking first for the correct type, and then simply accessing the `w_value` field.

```
guard_nonnull_class(p1, 12345, descr=<Guard0xabcdef123456>)
p2 = getfield_gc(p1, descr=<FieldP Cell.inst_w_value>)
```

For a constrained variable, the JIT will generate code to access the solver object (stored in the `inst_w_constraint` field), then retrieve the solver from that object, and finally call into the solver to get the actual value of the variable:

```
guard_nonnull_class(p1, 12345, descr=<Guard0xabcdef123456>)
p2 = getfield_gc(p1, descr=<FieldP ConstraintCell.inst_w_constraint>)
p3 = getfield_gc(p2, descr=<FieldP Constraint.inst_w_solver>)
guard_nonnull_class(p3, 56789, descr=<Guard0xabcdef456789>)
[... code to call into solver ...]
```

Thus, access to variables that do not use constraints is, after it is JIT-compiled, just as fast as in the base language without constraints.

Library-based Access

The library-based implementations trivially avoids any performance impact when constraints are not used. However, compared to a VM-based solution, we cannot rely on access to the JIT to optimize the case where variables do appear in constraints.

Our design still aims to provide good performance for a variety of applications that use constraints or access constrained variables. The Google Chrome's V8 engine allows inspecting the generated JIT code. The V8 JIT code is similar to RPython's. The main difference is that it also allows branches, rather than just checks for conditions.

In Babelsberg/JS, unconstrained variables are accessed directly, with just a check that the object layout has not changed:

```
CheckMaps t0 [0xabcdef123456]
t1 = LoadNamedField t0.field[in-object]@32
```

Just like in Babelsberg/R, the first line checks the type of the object pointed to by `t0`, and the second line reads a field from that object.

On the first access to a constrained variable, many more checks have to occur before we can call into the solver structure:

```
CheckMaps t0 [0xabcdef456789](stability-check)
t2 = LoadNamedField t0.current[backing-store]@40
Branch t2 goto (B10, B1) (Null,SpecObject) Tagged
B1:
CheckMaps t1 [0xabcdef123456,0xabcdef234567,0xabcdef345678]
t3 = LoadNamedField t0._isSolveable[in-object]@80
Branch t3 goto (B2, B11) (None) Tagged
B2:
t4 = LoadNamedField t2._definingSolver[in-object]@88
Branch t4 goto (B3, B12) (Null,SpecObject) Tagged
B3:
[... code to find variable value ...]
```

The first check tests that the global `Babelsberg` object has not changed. The second line is a branch that tests whether or not the `current` field on this global object is set. If it is, we are currently in constraint construction mode and must go to branch `B10`. In the case of an ordinary read, we instead check that the object we are reading from, `t1`, did not change its type, and then read the field `_isSolveable`. Again, we branch depending on the value of this field. If it were false, we can read the value directly using only one more access (branch `B11`). In this case the variable is solvable, and we must ask the relevant constraint solver for its value. To access the solver, we read the `_definingSolver` field and then enter solver-specific code to find the actual value.

Compared to Babelsberg/R, we have the additional check for CCM. This cannot be easily removed. Compared to RPython, V8 does not offer an interface for user code to give the hint to the JIT that CCM will only rarely be entered, and that we can deoptimize aggressively in that case. As in Babelsberg/R, reading a variable from the solver calls eventually calls `value` on the ConstrainedVariable in the property wrapper, instead of returning the original variable. We cache the value returned from the solver, and rely on the underlying VM to optimize read accesses when the cache is not invalidated.

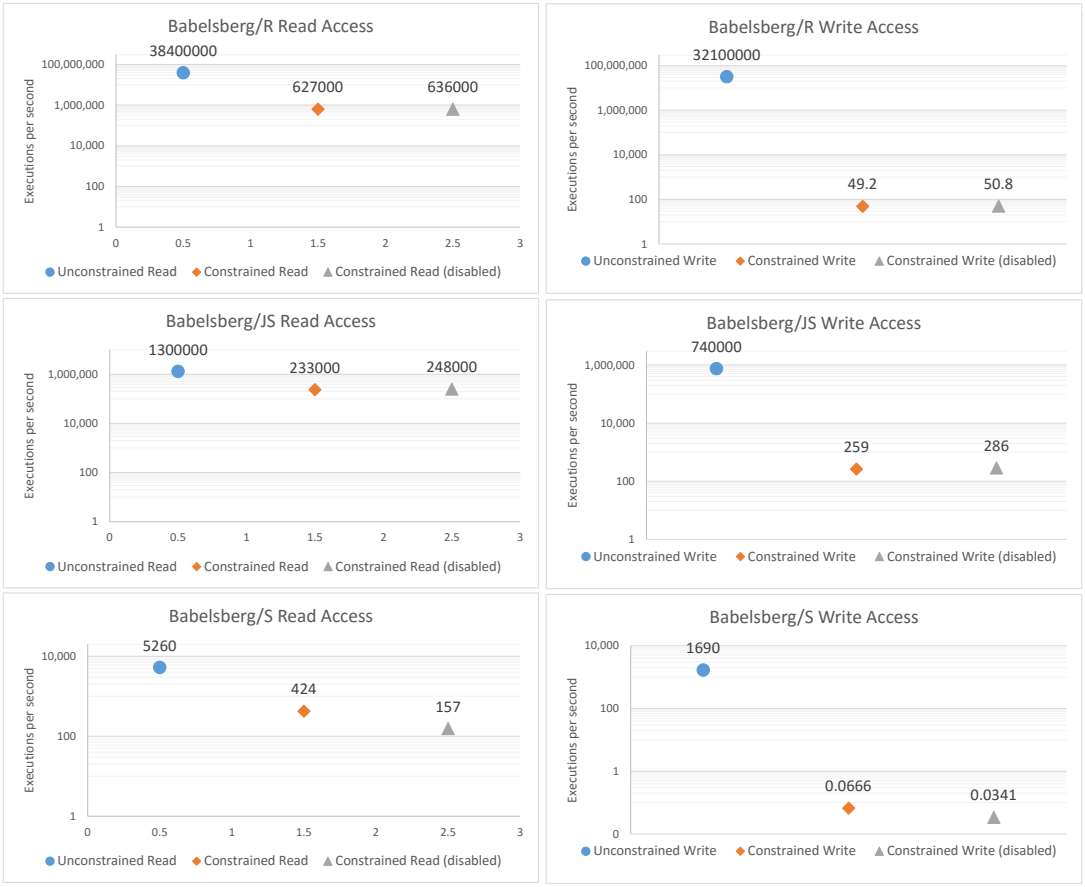


Figure 8.1.: Micro-benchmarks for read and write access to variables. All numbers show how many field accesses per second can be executed in the different scenarios (more is better).

Benchmark Results

To test how the purely object-oriented parts of a system are affected if we pass objects with constraints to them, we measured the overhead of field access for constrained versus unconstrained fields by repeatedly reading the same five properties from an object first without and then with equality constraints on each variable. In the benchmarks, the constraints are never violated, so we purely measure the overhead of accessing fields that have solver objects attached to them. For our benchmarks, we used the Cassowary solver, which for each of these systems is written in the host language and for which all three implementations are derived from the same Smalltalk code.

Figure 8.1 shows that the overheads for reading and writing to variables if constraints are present. To measure the read access, we averaged the runs over 500 repetitions of reading five different fields from an object. When we add a constraint, we require each of these variables to be equal to a fixed value. To measure the overhead of assigning without triggering the solver, we compare the time for unconstrained assignment to assignments to five fields that have a constraint `x >= 0` on them. In the benchmark, we simply increase the value if the variable 500 times in a loop. For both benchmarks we use the Cassowary constraint solver, which for all of our implementations is written in the host language itself. We have verified that in both cases, no solving takes place in the benchmarked loop, so the solver has little bearing on the results.

The overhead of intercepting variable lookup is very large in each of these cases. For reading variables our VM-based implementation in Babelsberg/R is within two orders of magnitude, whereas the library-based implementations remain within one order of magnitude. The reason is likely due to the optimizations of the tracing JIT in Babelsberg/R. It can optimize field access, but the additional steps required to read from constrained variables cannot be optimized well. In contrast, the method-based JIT compilers in Squeak and Chrome's V8 JavaScript engine experience comparatively less slowdown for accessing constrained variables. For writing variables, even if the constraint solver does not actually have to do anything to keep the constraints satisfied, many more methods are executed before the solver returns. Thus, the impact on writing variables is a slowdown of six orders of magnitude for Babelsberg/R and Babelsberg/S, and three orders of magnitude for Babelsberg/JS.

8.2. Performance Across Constraint-Languages

In this section, we evaluate the performance of the Babelsberg implementations and compare it with other languages that support constraint-programming. Turtle [53] and Kaplan [71] are used as examples of CIP languages; as well as on SWI-Prolog's CLP library [22], to compare the performance of constraint-imperative with constraint-logic programming.

Note that the benchmarks presented in Figure 8.1 can give an indication of the relative performance of Ruby, JavaScript, and Squeak on the run times used in these benchmarks: The Babelsberg/R VM executes field access about an order magnitude faster than Babelsberg/JS running on V8, and Babelsberg/S running on the Squeak Cog VM is about three orders of magnitude slower again than JavaScript. However, these micro-benchmarks cannot give a general indication of the performance of the host language implementations, but are important to keep in mind when comparing performance across Babelsberg implementations for constraint-programming tasks.

Comparing the run time of programs written to solve specific programming tasks across languages is difficult and often not meaningful. One option to compare the overhead of constraint solving between Babelsberg, Turtle, Kaplan, and Prolog CLP would be to use a set of classic OO benchmarks as a baseline for each language, and show the performance of solving constraint programming problems in relation to that baseline. However, we think such an approach would not produce meaningful numbers, because some of these languages call into external solvers whose performance has nothing to do with how well the language performs on OO benchmarks. The language runtimes and solvers used are also very different and their performance highly depends on the combinations of compilers, operating system, CPU architecture, and dependent libraries.

In our case, we compare the performance of solving programming tasks using constraints on very different languages with many different, performance-critical components. Rather than recommending one language over the other, this section attempts to give a general overview of what performance can be expected for each language in solving specific tasks. Unfortunately, good benchmarks to measure the intended use of languages that integrate constraints with imperative programming are difficult to find. Papers for such languages that evaluate performance use classical constraint solving problems. For our comparison, we selected three such problems: the *Send-More-Money* cryptarithmetic puzzle [25][1], a layout example from the Turtle distribution to proportionally position window panes (without preferential constraints, since these are not supported on Kaplan and Prolog), and the *Animals* puzzle found in some high-school textbooks[2]. We ran these benchmarks

[1] A student sends a telegram to her parents to ask for more money. The content of the telegram is SEND + MORE = MONEY. How much money does she ask for, if each letter encodes exactly one digit from zero to nine?

[2] Spend $100 to buy 100 pets. You must buy at least 1 of each pet. Dogs cost $25, cats $1, and mice $0.25.

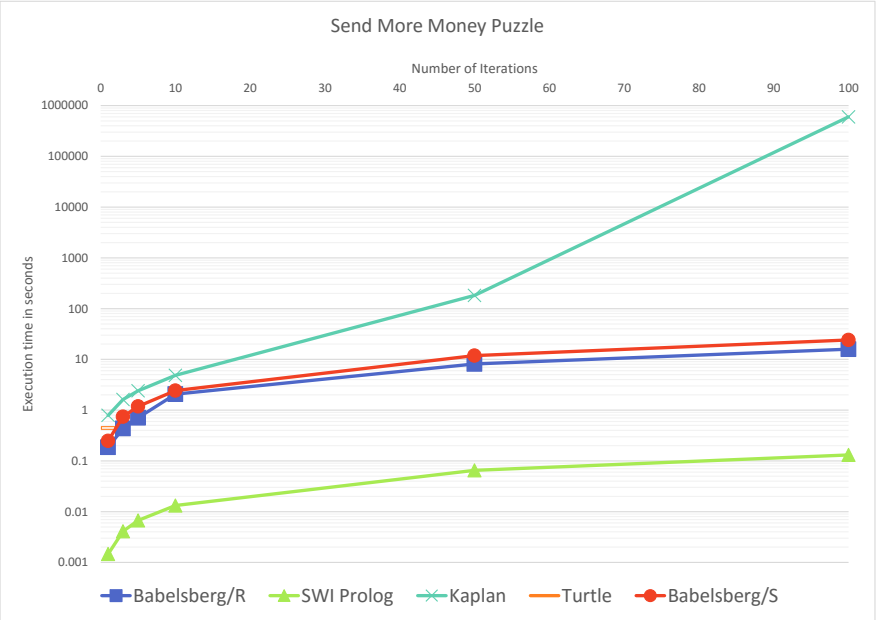


Figure 8.2.: "Send-More-Money" puzzle

on in loops of different length to allow the JITs to warm up, but also to measure how each language performs when dynamically adding constraints.

In these benchmarks, we use Turtle version 1.0.0. It was compiled using 64-bit gcc version 4.8.4. SWI Prolog was used in 64-bit version 6.6.4. Kaplan is implemented in Scala version 2.9.0.1. It was run on the 32-bit OpenJDK VM version 1.6.0.36. The 64-bit Java VM cannot be used due to the dependency on a custom built wrapper around the 32-bit version 4.4.0 of Z3. Each set of measurements was repeated ten times. We only include measurements that finished within 30 minutes. Error bars are not included, because all errors were below 5%. Execution time in seconds is given on the y-axis, the number of repetitions is on the x-axis. Lower y-axis values are better.

In the Send-More-Money puzzle (Figure 8.2), the Babelsberg/JS implementation never finished in the allotted time, and the Turtle implementation also seems to hang forever if we attempt to solve the puzzle more than once in the same process. For this clearly declarative puzzle Prolog is fastest, followed by Babelsberg/R, Babelsberg/S, and Kaplan at about two orders of magnitude slower. Kaplan, however, seems to have problems with dynamically adding constraints over many iterations and scales much worse.

In the Animals puzzle (Figure 8.3), we had to omit Turtle, because it never finished. Babelsberg/JS is clearly the slowest here, mostly because it uses a version of Z3 compiled into JavaScript and run in the browser for this benchmark, which the V8 JIT is bad at optimizing.

In the layouting example that came with the Turtle distribution, Turtle is also the fastest (Figure 8.4). While the performance of the other object-oriented languages is roughly the same relative to each other as in the previous benchmarks, Prolog is much slower here, only becoming faster than Kaplan for 100 iterations. Again, Kaplan seems to scale badly when dynamically adding many constraints.

Overall these benchmarks show that the CIP languages vary in their performance, and also that Prolog with a constraint satisfaction library is a better choice for these problems when no imperative features are used. In general, all implementations scale linearly. We observe that a primarily declarative language such as Prolog seems to still be the best choice when dealing with these predominantly

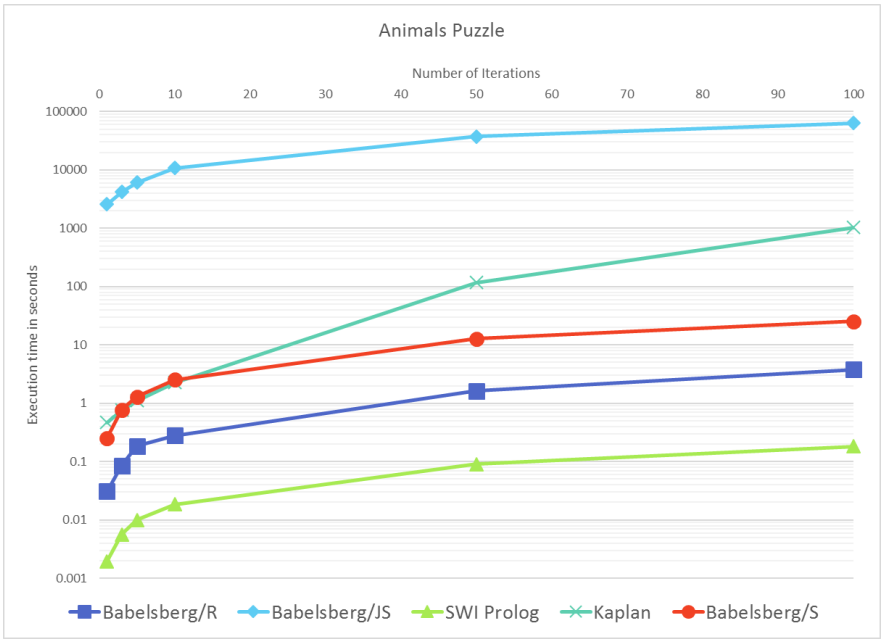


Figure 8.3.: "Animals" puzzle

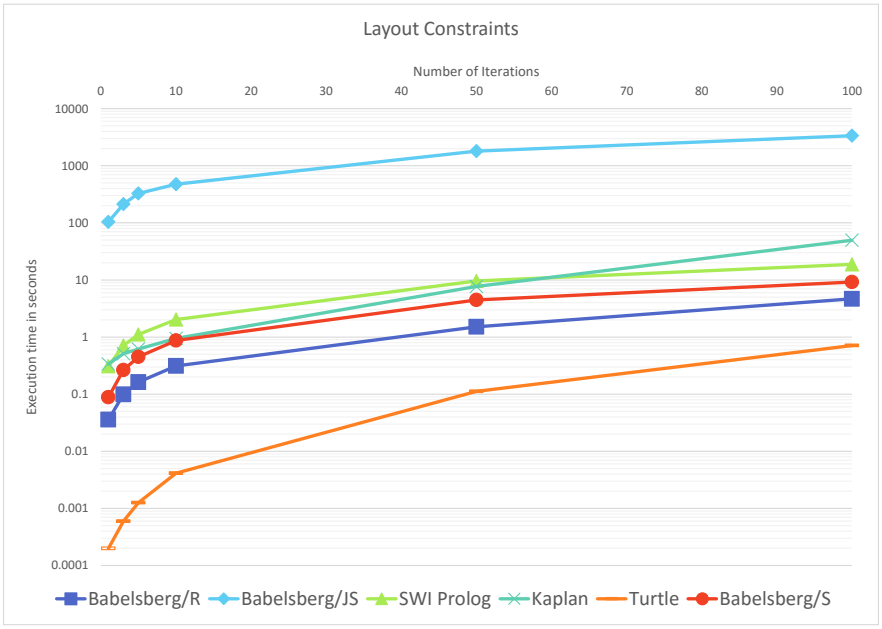


Figure 8.4.: Layout constraints

declarative problems. However, both Babelsberg/R and Babelsberg/S are feasible alternatives at only about one to two orders of magnitude slower than Prolog. We argue that this shows that the Babelsberg design can be implemented efficiently enough for selected use cases, both in-VM and as a library.

8.3. Fast Incremental Constraint Solving

Performance of the solving algorithm itself is important for Babelsberg languages. This is particularly apparent even when only a few variables change with high frequency and constraints have to re-satisfied in response, for example, when a graphical object should follow the mouse cursor. As described in Section 2.5, edit constraints are used to support incremental constraint satisfaction, and are important for achieving good performance in interactive applications that use constraints. The `edit` method provided as part of Babelsberg/R adds edit constraints and repeatedly updates them with values from a stream running in a separate thread. In Babelsberg/JS, 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. Babelsberg/S does not support edit constraints.

Since the language design supports cooperating solvers, the solvers have to provide a specific edit constraint API. If a variable is added to an edit constraint, but we dynamically discover that it is used in a solver which does not support the edit constraint API, a run-time exception is generated. 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.

resolveArray is used to supply 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.

finishEdit is sent to each solver variable when the edit stream ends or the callback is called without arguments. 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.

To use, for example, Cassowary as the solver, all edit variables must be of type float (e.g., the x and y values of a point), 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 methods may calculate values or just access fields). The system creates fresh 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.

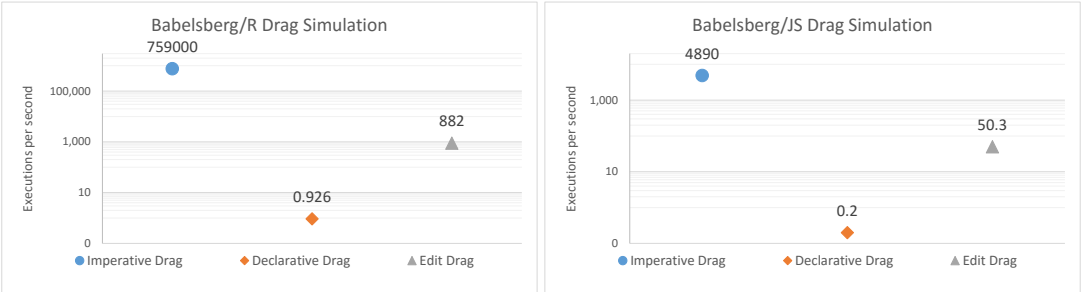


Figure 8.5.: A comparison of constraint solving performance for hand-coded imperative solving, Babelsberg-style solving through assignment, and Babelsberg with edit constraints. Numbers show how many re-solving operations can be executed per second (more is better).

In the following example in Babelsberg/R, 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.

Benchmark Results

To quantify the impact of edit constraints, we used an example from Kaleidoscope'93 [80] and adapted it to our Babelsberg implementations. In this example, the user drags the upper end of the liquid mercury in a thermometer using the mouse. However, the mercury cannot go outside the bounds of the thermometer, even if the user tries to drag it out. Additionally, a gray and white rectangle on the screen should be updated to reflect the new position of the thermometer liquid, and a displayed number should reflect the integer value of that position. Refactoring the imperative version for Babelsberg makes it more general, so the comparison is biased towards the imperative code. However, this example demonstrates the performance impact if an imperative program is refactored primarily with the goal to make it more readable, but not necessarily more general.

Note that the object-constraint version may be written in two ways: one that is more like the imperative version and assigns new mouse locations in a loop; and a more constraint-oriented version that declares `mouse.location_y` as an edit variable that triggers incrementally re-satisfying the constraints. The latter is expected to be much faster, as Cassowary can just re-optimize a previously optimal solution.

In this set of benchmarks we only include Babelsberg/R and Babelsberg/JS, since we did not implement edit constraints in Babelsberg/S.

For both implementations, the hand-coded imperative version is clearly the fastest. However, it is also the longest, hardest to understand, and most difficult to prove correct. It is also clear that the purely declarative versions are generally too slow to be used in an interactive application: in both cases, the system could re-satisfy constraints less than once per second, which is not enough to smoothly follow mouse movement and update the screen. Although using edit constraints is

still an two to three orders of magnitude slower than pure imperative code, it is fast enough to re-satisfy constraints in an interactive application and still provide smooth display updates.

8.4. Automatic Edit Constraints

We have shown in Section 8.3 that edit constraints can significantly improve the performance of constraint solving and in some cases make the use of constraint solving feasible in the first place. However, using edit constraints directly requires developers to know about them, understand where they are useful, and to adapt the source code to create edit constraints.

Automatic edit constraints (like our integration of state and object-oriented behavior with constraint declaration, or the architecture for cooperating solvers and solver selection heuristics) support the goal of Babelsberg to make constraints a useful tool for developers not familiar with constraint solvers. Rather than requiring OO developers to learn about incremental solvers, in Babelsberg we have added heuristics for two solvers, DeltaBlue and Cassowary, to recognize variables that change frequently and use automatically apply incremental re-satisfaction to speed up solving.

The solving process of DeltaBlue when a variable is assigned involves creating a new equality constraint, creating an execution plan, executing that plan to solve the constraints, and finally to remove the equality constraint and the associated plan. As a simple optimization heuristic, we can keep the last n equality constraints and plans in a cache; if one of the n variables changes again, we can just update the equality constraint and re-use the execution plan, rather than re-creating it.

For Cassowary, the simplex tableau has to be prepared for incremental re-solving, and all variables that will be assigned have to be known beforehand. This preparation is potentially slow, and it is not possible with the implementation of the algorithm to keep multiple optimized versions of the tableau around. Thus, the heuristic we use for Cassowary is more akin to a JIT: for each variable known to Cassowary, we keep a counter that tracks how often the variable has been assigned recently. In regular intervals, we check which n variables were assigned to most frequently, and optimize the tableau for changes coming from those variables. At the same time, we decay the counters either by a fixed percentage or value. This ensures that variables that have been assigned to often in the past, but not much in the recent history of the program, are no longer considered for incremental re-satisfaction.

Benchmark Results

We present two sets of benchmarks, one each for DeltaBlue and Cassowary. In our benchmarks, we measure a chain of variables that should be equal to a fixed sum, a horizontal drag of a slider where only one variable changes, a mouse drag where two variables changes alternate, a mouse drag where one dimension changes more frequently than the other, and finally a mouse drag where each dimension changes five times, and then the other changes five times.

For DeltaBlue, the best heuristic is to keep the last edit constraint in a strategy we call *last JIT*. At any time, if there is no current edit constraint, we create one. If the current edit constraint is for another variable, we throw the old one away and create the new one. This implementation could easily be extended to cache the last n edit constraints, rather than just the last one. However, even as it is, we can see in our benchmark that this heuristic is almost always as fast as writing edit constraints directly, and is never slower than not using edit constraints at all.

For Cassowary, we include three heuristics with Babelsberg: the *classic JIT* strategy decays counters and changes or creates edit constraints in fixed intervals. A second strategy, *additive adaptive*, increments the interval any time the most frequently used variable did not change, and decrements

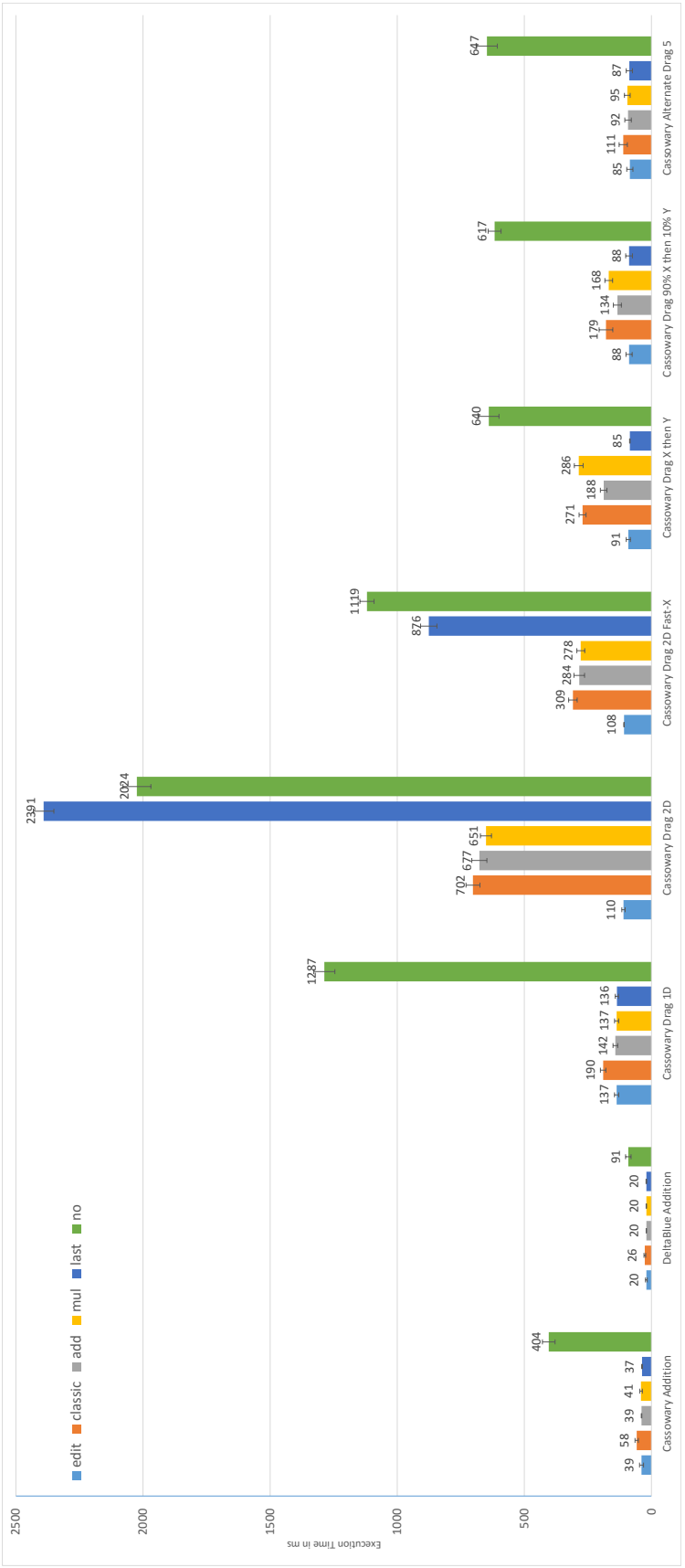


Figure 8.6.: Combinations of benchmarks and automatic edit constraint JITs. Graph shows execution time required for 500 solving operations (less is better).

the interval any time it did. Thus, when an edit constraint is well chosen and can stay in effect for a long time, we reduce the overhead of checking and decaying counters linearly with time. A third strategy, *multiplicative adaptive*, modifies this behavior and, rather than incrementing or decrementing the interval, it multiplies or divides the interval by two. This has the advantage that we can avoid re-checks for longer periods of time if the edit constraint stays in effect.

Our benchmarks show that all our heuristics are faster than using no edit constraints at all. The adaptive strategies generally work better than the *classic JIT* strategy, and by default we use the *additive adaptive* strategy, but the developer can select a different strategy manually.

Summary

Our benchmarks illustrate that the performance penalty of using constraints intermingled with ordinary object-oriented code is very large. This issue is particularly relevant if we spent a lot of time in pure object-oriented code that only reads from many constrained variables. It is less of a problem for code that reads a few variables, but spends the majority of time on calculations or IO. The overhead is also less relevant for applications that have constraint solving at its core, since our benchmarks also show that the Babelsberg prototypes compare favorably to other constraint-imperative programming systems. Nonetheless, there is potential for improvement. Making use of edit constraints can significantly improve performance, but only for solvers which support them. Adding additional optimization strategies for other types of constraint solvers is part of our future work.

Part IV.

Applications with Object-Constraint Programming

9. Using Constraints in Object-oriented Applications

Using our prototype implementations of the Babelsberg design, we have written or extended a number of applications, including some from the domains of load balancing, electrical and mechanical simulation, puzzle solving, unit conversion, games, and layouting. We have also evaluated applications written by 24 graduate students using Babelsberg that did not have any prior experience with Babelsberg or Object-Constraint Programming (OCP) in general. These applications were written with object-oriented principles in mind (using the idioms and patterns common for the underlying host languages) and employ constraints where it simplifies the code.

In this chapter, we discuss three usage patterns and issues that emerged in some of these applications. Due to the limited number of applications, we cannot generalize these to give recommendations on how to use Babelsberg and further research into that area is required.

9.1. Constraints at Initialization

A number of our applications use constraints to declare some properties that objects must adhere to. This is useful, for example, in simulations of physical circuits, where each physical part (wires, resistors, ammeters, etc.) should follow the relevant physical laws (Ohm's law, Kirchhoff's current law, and so forth).

Figure 9.1 shows a constraint-based simulation of Wheatstone Bridge being constructed in the Lively Kernel environment using Babelsberg/JS. (A Wheatstone Bridge is used to measure an unknown electrical resistance by balancing two pairs of resistors so that the electrical potential between

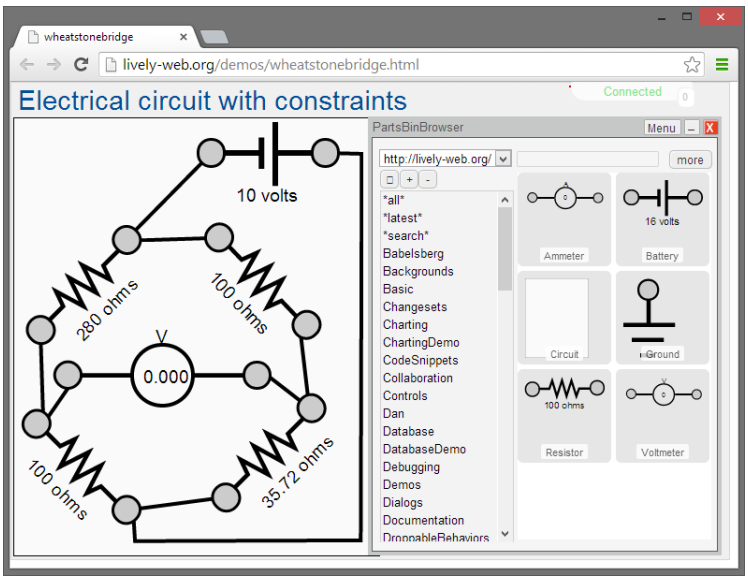


Figure 9.1.: Constructing a constraint-based Wheatstone bridge simulation

them is zero.) Graphical objects ("Lively Parts") representing batteries, resistors, and meters are copied from the Lively Kernel Parts Bin [77] on the right, dropped into the circuit on the left, and wired together. The circuit parts are represented by classes that create constraints on initialization. For example, any object that is connected via two endpoints inherits from `TwoLeadedObject`:

```
class TwoLeadedObject {
  constructor() {
    this.wire1 = {voltage: 0.0, current: 0.0};
    this.wire2 = {voltage: 0.0, current: 0.0};
    always: { this.wire1.current + this.wire2.current == 0.0 }
  }
}
```

This constraint represents Kirchhoff's current Law that the sum of currents for the inputs and outputs must be zero. Other components such as resistors inherit from `TwoLeadedObject`. Calling the inherited implementation works as expected, that is, it creates the constraints on the subclass' instance. Any additional, subclass-specific constraints are added after that:

```
class Resistor extends TwoLeadedObject {
  constructor(resistance) {
    super();
    this.resistance = resistance;
    always {
      this.wire2.voltage - this.wire1.voltage ==
          this.wire2.current * ro(resistance)
    }
  }
}
```

Whenever a new object is created, it automatically follows the physical laws as far as they are represented in the constraints (setting all voltages and currents to zero). To connect two components, we simply constrain the connected wires to have the same voltage, and the sum of the currents to be equal to zero. Thus, if we connect a battery component (with a non-zero voltage potential between the two wires), the system automatically solves the constraints when the parts are first connected, and re-solves them if, for example, the battery's supply voltage or a resistance is edited, updating the voltage displayed by a meter.

9.2. Dynamic Argument Checking versus Constraints

One issue we encountered is that many JavaScript methods have, besides a return statement, some statements that check the number, types, or structure of arguments and just throw an error if the arguments are incorrect. As per our semantics, such methods do not work multi-directionally. In fact, allowing them to be truly multi-directional would allow the solver to change the number, types, or structure of arguments, which is the kind of surprising behavior we want to avoid. As an example, consider the frequently used method to add two points in Lively Kernel:

```
function addPt(p) {
  if (arguments.length != 1) throw ('addPt() only takes 1 parameter.');
  return new lively.Point(this.x + p.x, this.y + p.y);
}
```

We almost certainly do not want to satisfy a constraint that calls the `addPt` method with a bad number of argument by dropping an argument:

```
always: { point1.addPt(point2, bogus).eqPt(point3) }
```

Instead, the argument check should not be part of the constraint. During the development of the Babelsberg design, we initially allowed such methods in constraints that simply do argument checking in this manner. The argument checking code was simply executed once and then ignored during constraint solving. (We have published multiple applications that use such methods as a reviewed software artifact [34].) These methods worked as expected as long as we did not include constraint solving procedures that can solve for the length of arrays, because the arguments array would become a part of the constraints, but could never be solved for.

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

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

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

In our design and semantics, the specifics of the solver no longer influence the meaning of using methods with branches or argument checking — it can only be used in the forward direction (cf. Section 3.2.5). As a workaround for using it multi-directionally, the developer must move the tests outside of the method and add the constraint only for the branch that is chosen.

In future work, we will investigate how to add a mechanism to recognize this pattern and automatically mark the branch condition as read-only to the solver. We considered allowing the developer to manually mark such statements to be ignored during constraint construction mode. However, that makes it impossible to ensure that the semantics of the method during imperative execution matches the semantics of the translated constraints in the solver. In the extreme this could lead to situations where the solver correctly satisfies its constraints, but the return value of the constraint expression in imperative mode is still false. This actually happened in an experiment with such an annotation, where we introduced a bug in our branch condition, and would always throw an exception, even though the solver would say it had satisfied the constraints. Such failures are then very hard to debug, since the state looks correct, printing of debugging information does not show the correct constraints, and only reading the code can reveal the bug.

9.3. Local Variables for Read-Only Computation

Our implementations of Babelsberg include methods to mark variables as read-only for the solver. However, read-only annotations do not have a clear semantics, and thus cannot be used in imperative code. This is a problem when writing methods that should be usable both in ordinary code as well as in constraints, but parts of which should be read only for the solver.

However, we have found that many of our students moved computation that should be read-only into separate methods, used local variables to split up the computation, and thus ensured that such methods could only be used in the forward-direction. One such method we found calculates the pressure of an object representing a balloon filled with some gas:

```
class Balloon {
  entropyPerVolPromille() {
    var gasConstant = this.gasConstant, // J/(kg*K)
        density = this.density; // kg/m^3
    return gasConstant * density / 1000; // J/(K*m^3*1000)
  }

  pressure() {
    return this.entropyPerVolPromille() * this.K; // kPa
  }

  constructor(temperature, density, gas) {
    this.K = temperature;
    this.density = density;
    this.gasConstant = this.lookupGasConstant(gas);
    always: { this.pressure() <= 115/*kPa*/ }
  }

  ...
}
```

The desired function of the constraint is to restrict the temperature (in Kelvin) of the object to remain such that the pressure remain below 115 kPa. The code first assigns the gas-specific constants required to calculate pressure to locals, so that the constraint solver will not be allowed to modify these constants on the object to satisfy the constraint. Here, it would be inconvenient to use a read-only annotation. If used inside the pressure method, the semantics of that method when used in ordinary imperative code are not clear. The other option would have been to manually inline the pressure method and add a read-only annotations only on the return value of the `entropyPerVolPromille` method, but we would like to avoid this code duplication.

Summary

We have presented three usage patterns and the most common issue that can be found throughout our and our students' example applications. This work is young, and we expect to find more common patterns and issues as we explore it further. However, we have found that becoming aware of the above cases already helped us structure our applications. Both declaring constraints at initialization time, as well as explicitly creating methods that work only in the forward direction are so common that we are investigating if there should be support for these patterns in the language. Regarding the issue of argument checking and branching, we are experimenting with an extension

to our design that is similar to our extension for allowing collection predicates in constraints. To that end, we need to identify the most common patterns of such branches and their expected effect.

10. Constrained Scopes, Behavior, and Events

In this chapter, we present a larger applications, a clone of the game *Wii Play/Tanks!*[1] that runs in the browser. The applications in the previous chapter were comparatively small, allowing us to focus on details of the usage patterns. Using this larger example application, we discuss how to modularize and group constraints and show interesting examples of how constraints can integrate with other language extensions.

Figure 10.1 shows the running game. It is top-down, with the players controlling toy-tanks that shoot rubber bullets at each other. The tanks can move about the field and turn their turrets independently. The field contains obstacles such as walls and holes in the ground which the tanks cannot cross, but bullets can ricochet off walls and fly over holes. As an extension to the original, we added power-ups which give the tanks' bullets additional behavior such as flying faster or ricocheting more often.

Context-oriented Programming (COP) [18, 55] is a mechanism to dynamically adapt behavior based on layers. From our experience with both OCP and COP we realized there are multiple opportunities to use constraints and layer-based behavioral adaptation. Power-ups change the behavior of bullets, and, during development, alternative drawing modes are desirable for introspection and editing. These can be conveniently encapsulated using layers. Additionally, there are UI elements that have constraints on them, such as the target following the mouse, or that a tank cannot drive into a wall or over a hole. We encapsulated these using constraints. During the implementation of this game, we have used constraints as an activation mechanism for COP layers, calling this mechanism *Layer Activator Constraints*. Conversely, we have included a mechanism for layers to add and retract constraints, called *Constraint Layers*.

[1] ©2007 by Nintendo®, `http://www.nintendo.com/sites/software_wiiplay.jsp`, accessed March 30, 2015

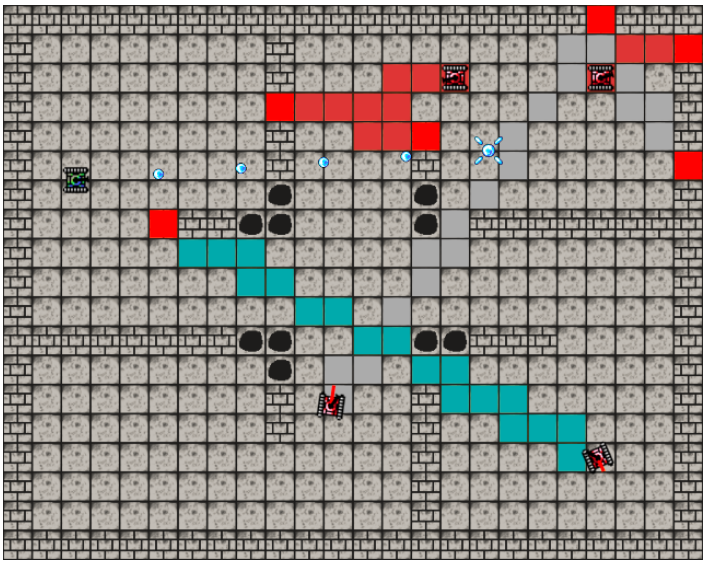


Figure 10.1.: "WePlayTanks" with Babelsberg and ContextJS

10.1. Layer Activator Constraints

There are various activation mechanisms in COP languages, including dynamically scoped [55] or scoped to object structure [76] and event-based [1, 70] or implicit activation [81]. While the first two mechanisms offer explicit control over the activation time of layers, the latter two are particularly interesting for our discussion because of their declarative nature.

Event-based activation uses system-generated events to activate or deactivate a layer during the event loop. Reactive activation is based on similar principles as Aspect-oriented Programming (AOP): whenever a message is sent that matches a *pointcut*, an associated condition is evaluated. If that condition evaluates to true, a layer is activated for the duration of the method activation.

A limiting factor of both mechanisms is their granularity. Event-based activation relies on the system to generate events that indicate a change in the system state. It is generally not possible to tell the system to generate an event whenever an arbitrary condition changes. Similarly, reactive activation focuses on message sends. The conditions for layer activation are only checked at join-points, and thus the burden is on the programmer to ensure that their pointcuts match all message sends that may be relevant. With Babelsberg, however, we can formulate a constraint that connects the activation state of a layer to an expression:

```
always: {
  game.time < powerUp.creationTime + powerUp.timeout ==
     powerUpLayer.isActive()
}
```

This constraint is used in the game to change the behavior of the game as long as the `powerUp` is active, that is, as long as its `creationTime` and `timeout` are larger than the current game time. This uses the standard Babelsberg/JS mechanism for constraint solving: the predicate is interpreted to determine its dependencies, and the objects that affect the outcome of the predicate are wrapped to monitor changes to them.

As part of the game we included a custom constraint solver that can deal with COP layers, implemented using our cooperating solvers architecture. When this solver is called to satisfy the above constraint it adjusts the activation state of `aLayer` to match the result of `my.condition()`.

Since Layer Activator Constraints are built around a shared constraint solving mechanism, the semantics for competing or contradictory activators arise out of the theory of constraint solving for contradictory or competing constraints. If two activators contradict each other, solving will fail with an exception at the time the second activator is defined — with the effect that the second activator is not enabled.

10.2. Constraint Layers

Constraint programming typically revolves around describing a global system state, which is why the global activation mechanism provided by `always` and `once` are sufficient. OOP, on the other hand, focuses on defining independent modules and often deals with changing states.

In our game, consider that the turret of the tank should follow the mouse cursor controlled by the player:

```
always: {
  player.turretDirection.equals(input.mouse.sub(player.position))
}
```

This describes a global assumption, which will be true when we are playing. However, the game consists of various additional states, such as an edit mode. Analogous to partial classes and their behavior in behavioral layers, for this game, we introduced scoped constraints in *Constraint Layers*.

COP layers as implemented in ContextJS [76], a JavaScript implementation of COP, are first class objects that *refine* classes. A refinement that is active when playing our game is defined as follows:

```
GameLayer.refineClass(
  Game, {
  timeout: function () {
    cop.proceed();
    this.resetTimer();
    this.restartLevel();
  }
})
```

Here, the `GameLayer` adds the `timeout` function to the `Game` class. When the layer is active, instances of `Game` respond to the timeout function by first calling the base implementation (line 3), and then resetting a timer and restarting the level (lines 4–5). This definition is, in COP parlance, called *partial behavior*.

Constraint Layers can add constraints to layers in the same way as partial behavior. Like partial behavior definitions, scoped constraints are enabled when the associated layer becomes active and are retracted when the layer is deactivated:

```
GameLayer.addConstraint(
  always: {
    player.turretDirection.equals(input.mouse.sub(player.position))
  }
);
```

This extension in particular seems useful in broader contexts to modularize the use of constraints. As part of future work, we are planning to formalize this idea and integrate it with our semantics.

10.3. Constraint Triggers for Reactive Behavior

Reactive programming is used to trigger certain behavior in response to events. Functional reactive programming languages allow a lot of freedom in what kinds of expressions can be used as event sources. We have found that using Babelsberg, we can offer a similar mechanism.

In the game, bullets shot from the tank should ricochet off walls. In functional reactive programming, we would say that a bullet moves along a vector, and when it intersects a wall, its vector is reflected. We could implement a solver for reflecting bullets off walls, and hand it the following constraint:

```
always: {
  !bullet.intersects(wall)
}
```

In this case, the solver should satisfy the constraint by reflecting the bullet and moving it out of the wall.

We have found a more general solution in form of a custom, configurable solver that used user-defined functions and local propagation. This solver is initialized with a callback, and then used explicitly in constraints:

```
var solver = new ReactiveSolver(function () {
  bullet.reflectOff(wall);
  bullet.moveForward();
});
always: {
  solver: solver;
  !bullet.intersects(wall)
}
```

Here, we use the Babelsberg/JS syntax to pass a solver explicitly. When this instance of the reactive solver is called to re-satisfy the constraint, it executes our callback. (The developer has to ensure that the callback actually does satisfy the constraint, just like in other configurable local propagation solvers.)

Finally, bullets should only ricochet of walls a fixed number of times, and then vanish. The `ReactiveSolver` can take two more optional arguments. The first is a counter to decrement anytime the solving callback is executed. When the counter reaches zero the third argument, a cleanup function, is called, and all constraints in the solver are deactivated:

```
var solver = new ReactiveSolver(function() {
  bullet.reflectOff(wall);
  bullet.moveForward();
}, 2, function() { bullet.remove() });
```

This extension demonstrates how functional reactive programming is contained in OCP, and also shows a method of scoping constraints in time.

Summary

In this larger application, we found that our design lacks fine-grained control over the scope of constraints, as well as a mechanism to group constraints into modules and re-use them in different contexts of the program. We have no principled answer to these issue, yet, but our experience suggests that the combination with Context-oriented Programming has useful properties that a principled solution should also bring to our design, such as a clean way to enable and disable multiple constraints at once.

Another interesting observation is how our architecture for adding multiple solvers can be used to add special purpose solvers for activating layers or implementing reactive behavior. This suggests that our architecture may in fact be more general than just a design for cooperating constraint solvers. Instead, it could be used to combine general problem solving approaches. As part of future work, we are investigating ways to combine, for example, Prolog-style backtracking with Babelsberg.

11. Debugging and Understanding Constraints

A secondary goal of the Babelsberg design is to improve programmer productivity by allowing developers to use constraints instead of object-oriented code where the former are more concise, expressive, or maintainable. In general, however, more time is spent understanding and debugging programs than writing them. To improve programmer productivity, providing adequate debugging tools is a thus key ingredient, as without them, debugging is a manual, time-consuming task [24]. In this chapter we present tools that we developed to help us to understand, debug, and experiment with constraints in Babelsberg.

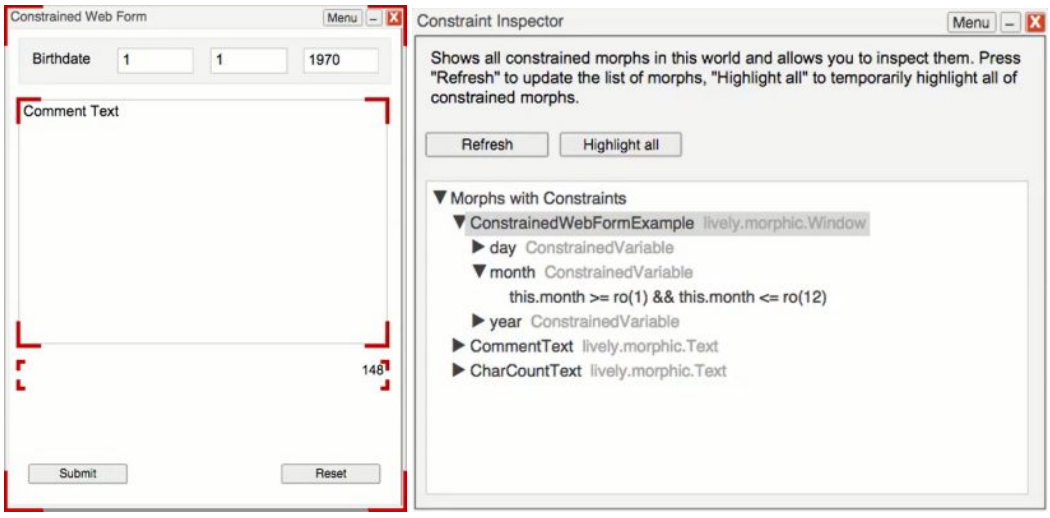
The tools we present are built for the library-based implementations of Babelsberg. They take advantage of the fact that all meta-object structures are implemented in the host language, and can thus be easily accessed from user code.

11.1. Inspection

Especially in live programming environments such as Squeak and the Lively Kernel, the life-time of constraints and which objects they affect can be hard to follow. Both Squeak and Lively use a graphical environment based on Morphic [85], with direct manipulation of objects in a scene-graph. In such a live environment, we want to easily get answers to a number of questions: Are there already constraints on graphical objects in the world, and if yes, which objects are involved? What constraints are on one particular object? What strength is attached to a constraint, what solver is it in, and is it currently enabled or disabled?

Lively Kernel, Squeak, and similar environments come with *object inspectors*. These tools enable direct observation of program state, allowing the developer to systematically check for unexpected states, by drilling down into an object and observing how fields change when interacting with it. These inspectors also allow developers to run code in the context of an object and to change the values stored in fields.

Since the observation of facts is useful during debugging and often used as a first approach, we have created a *constraint inspector*. Constraints are global, usually connect multiple objects, and can interact or possibly contradict one another. Consequently, this inspector does not only show one constraint, but instead tracks all constraints and how they relate to graphical objects. Figure 11.1 shows our inspector and a graphical data entry form with constraints on it. The inspector shows a list of Morphs that have constraints on them and highlights these. We can also use the inspector to observe if and how an application dynamically adds constraints as we interact with it. We can explore the constrained morphs to find which parts of them contribute to constraints. In the example, the input fields to enter a date of birth have constraints on them to ensure they cannot have invalid content. We can see the constraint expression of a constraint on the `month` field. Double clicking this expressions allows us to see and change the constraint's strength or enable and disable it for testing purposes.



(a) A constrained Web form (b) Constraint inspector

Figure 11.1.: The inspector highlights the graphical parts of the Web form that have constraints on them, and allows drilling down into the fields and constraint expressions on them.

11.2. Intercession

Besides logging or inspecting state, the most common tools for finding bugs in programs are debuggers. The Squeak/Smalltalk debugger in particular is very flexible and allows step-wise access to program states, inspecting the stack, as well as changing variables or program code at almost any point during the execution. Changed variables and code can be used immediately to test how the program behaves if it continues running with the changes.

Figure 11.2 shows how we extended the Squeak debugger when stepping through a 2D point equality constraint with an additional pane on the top right ① that shows the currently active constraints, and a special inspector on the right hand side ② showing the constraint that is currently being created. The debugger works as it normally would when running imperative code. However, when we enter constraint construction mode, the debugger additionally tracks the constraints as they are created. In the below example, we assert that `pt1` and `pt2` should be equal (③). From just looking at the expression we cannot tell how many constraints would be created from this expression. We could infer from the implementation if the = method for 2D points that we will create two constraints, one for each pair of dimensions on those two points. Our debugger also allows us to observe this fact and displays the equations that have been translated for the Cassowary solver in the top right pane. On the right hand side, we can see the details of the first constraint and for example change its strength or the generated expression to see how that changes the program behavior. Additionally, we can step into the procedure that assigns updated values from the constraint solver to the program variables and thus see the global effects of a constraint. This is particularly useful to understand which solution a solver chooses for a particular constraints and how many variables are changed in which way.

11.3. Experimentation

We have combined the tracking of variables, constraints, and graphical objects into an interactive *playground* to explore constraints and their effects. This playground offers developers an environ-

Figure 11.2.: The Babelsberg/S debugger

ment where graphical objects that have constraints on them can be safely manipulated, using the Lively Kernel's tools. One issue with a self-supporting development environment such as Lively, which this addresses, is that it can be hard to keep the environment and the application sufficiently separated to avoid that experiments with the application accidentally break the tools [75].

Figure 11.3 shows a small electrical circuit simulation with a battery and a voltmeter. The playground with graphical objects ① explicitly separates the tools from the application. All graphical objects are continuously monitored by the value tracker ②. This tracker shows all variables in the application that are determined by a constraint solver and their value. As we interact with the application, we can observe which variables are updated to re-satisfy constraints. In the code pane on the lower right ③ we can edit the code, both imperative and declarative, that makes up the application. The editor offers more than simple code editing:

- On hover, the variables that a constraint refers to are highlighted in the value tracker.
- Each constraint shows a checkbox to activate or deactivate it, which also updates the application and the value tracker.
- For each constraint, the currently selected solver is shown, and on hover a drop down list of available solver allows us to experiment and review the results that different solvers would produce.
- On right click, a list of priorities allows the developer to directly change the strength of the constraint.
- All numeric values are "scrubbable", that is, they can be manipulated by hovering over them and scrolling with the mouse, and the effects of the changed constraints can be observed immediately. This is useful, for example, in layouts where values like margins must be chosen to fit aesthetic preferences.

This playground has proven very useful to understand which variables are related by a particular constraint and to understand which solvers produce better solutions for a particular domain (e.g., that the lack of precision in relaxation solvers is no problem in layouting applications, but is not acceptable for circuit simulations). This helps understand the trade-offs involved in the heuristics for automatic solver selection as presented in Section 5.3.

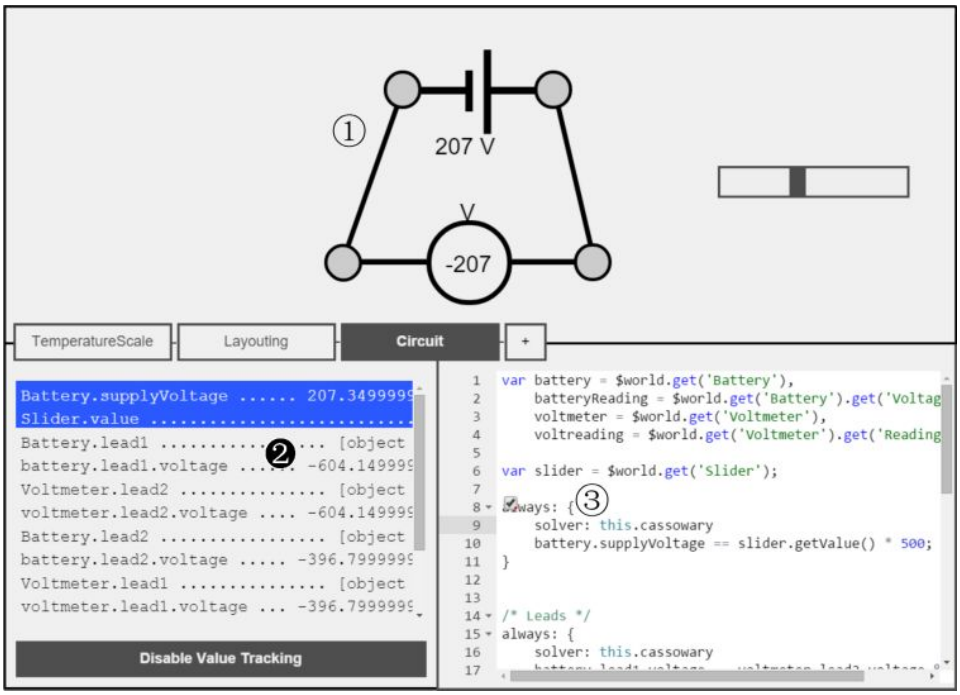


Figure 11.3.: A Babelsberg/JS playground

Summary

Our tools have already proven useful when developing Babelsberg applications, although without any alternative tools to compare them to, this says very little about their quality. It is clear that more work is required to understand which additional problems these tools need to address, where they could be made better, and how they should be integrated into the development process.

Part V.

Discussion and Conclusion

12. Related Work

In this chapter, we present the related work on using constraint programming systems. The chapter groups the presented systems by how derive from each other or some common previous systems, starting with purely constraint-based systems, over multi-paradigm languages for Constraint-Logic Programming and Constraint-Imperative Programming, to more recent data-flow constraints (often used in functional-reactive programming systems), and purely library- or DSL-based constraint programming. We think this grouping is useful in talking about some of the design decisions and trade-offs that set these systems apart from Babelsberg-style Object-Constraint Programming languages. At the end of this chapter we will also describe some design dimensions which we think are useful to consider with these systems and thus provide a different classification of them in relation to the work presented here.

12.1. Constraint Programming Systems

One of the earliest examples of a programming system that makes use of constraints was *Sketchpad* [115]. Besides pioneering computer-aided design, object-oriented programming, graphical user interfaces, and demonstrating the use of pens as a novel human-computer interaction technique, it is also an example of a system that integrates different constraint solving algorithms into the system to solve problems that may be too difficult for either one or the other solver. Starting with basic lines and arcs, Sketchpad could apply a set of predefined constraints to them, such as making lines parallel or perpendicular. Shapes could also be duplicated and composed, with the original shapes becoming master copies. Any further modifications to the masters were propagated to the copies.

Sketchpad includes two solvers, a one-pass solver and an incremental method. The first is a local propagation solver that propagates degrees of freedom. This is described as the preferred method by Sutherland for much the same reasons we have given in Section 2.3: if it can be applied, it is both simple and efficient. This method could very often be used in Sketchpad when constraints were defined with read-only variables (called "reference-only"), because that limited the degrees of freedom and provided ordering for a particular constraint. However, Sketchpadrecognized the need for simultaneous solving in multiple directions at once, and thus included also an iterative numerical solver that linearizes the constraints and then tries to minimize the errors. (We have integrated this method into the work presented here as "Sketchpad's Relaxation Solver".) The system also used a third solver not explicitly described in the thesis, a simple local propagation solver that propagates known values directly, before attempting to use propagation of degrees of freedom [7]. In contrast to the work presented here, Sketchpad integrated a fixed set of solvers, and users thus had to be careful to add constraints in a way that would allow the system to use one of these [115, p.112ff].

Another early example of a constraint-based system used *finite domain constraints* to interpret two-dimensional drawings of three-dimensional scenes with shadows [120]. This system includes a vocabulary of possible line intersections that can be present in such drawings. These are assigned to the drawings as labels, and each label adds constraints on how what labels can be at connected lines. The system then searches for matches of intersection types that describe a valid three-dimensional

structure. Compared to this work, this early use of constraint solving is very specific to the problem domain. The generated interpretations are concise because they can omit many implicit constraints that are specific to the domain, a feature that in some way we have attempted to emulate with implicitly generated stay constraints and our principled limitations on constraints over object structures and identity.

ThingLab was a constraint-oriented simulation laboratory implemented as an extension to Smalltalk-76 [64]. Although not meant as a general-purpose programming tool, it did allow the definition of constraints on objects' parts, and those constraint definitions could be inherited and composed in an object-oriented manner. ThingLab had many similarities to Sketchpad including the use of the same three solving mechanisms [7], and the graphical construction of objects and object-compositions. Its integration with the Smalltalk programming language, however, made it easier to create custom constraint-satisfaction methods that could be used for local propagation. ThingLab also pioneered some uses of constraints for simulations, animation, unit conversions, and spreadsheet programming, and we have re-created some of these applications as an evaluation of the work presented here. ThingLab also optimized solving performance by compiling Smalltalk methods that implement the re-satisfaction plan for constraints given a change to a particular field. A similar mechanism could be applied to our work to improve the performance for some solvers that do not support edit constraints.

12.2. Constraint Programming in General Purpose Languages

Attempts to integrate constraints with general purpose programming have a long research history. One of the first systems to support constraints within a general-purpose programming paradigm is the Constraint-Logic Programming scheme [68], which evolved from logic programming. CLP can be seen as a generalization of logic programming, where the constraint domain of pure logic languages is the set of terms available in that language. Jaffar and Lassez introduced a family of CLP(X) languages, where X stands for the domain on which the constraints in the language work. One instance, *CLP($\mathcal{R}$)* [69], provides constraints over real numbers in Prolog. Another language of this kind is Concurrent Constraint Programming [110]. These languages are in the logic programming family, and in their standard form have no notion of state or state change, in contrast to the work presented here. CLP languages have significant advantages, such as a clean semantics, but they sacrifice the familiar capabilities and programming style of the more mainstream imperative paradigm. In contrast, our goal in this work was to support an imperative, object-oriented programming style that integrates constraints syntactically and semantically.

The *Kaleidoscope* language is a first example of a Constraint-Imperative Programming language [46] that aims to integrate constraint solving with the imperative paradigm. Kaleidoscope supports Smalltalk-like classes and instances, and in addition, integrated constraints with the language itself. This integration included built-in constraints over primitive objects (such as floats) and constraints over user-defined objects, which were provided by *constraint constructors*. For example, the + constraint for Points could be defined using a constraint constructor `a+b=c` that expanded into constraints on the `x` and `y` instance variables of the three points `a`, `b`, and `c`. Separately, the language also provided OO methods. Both constraint constructors and OO methods were selected using multi-method semantics. This accommodated, for example, the case of a constraint constructor call `a+b=c` in which `b` and `c` were known and `a` was unknown. In contrast to the work presented here, were only "ordinary" imperative methods exist and are used both in constraints and imperative code, Kaleidoscope keeps the abstractions for constraint solving and object-oriented behavior separate.

Another early CIP systems is *Siri* [58]. This work, inspired by ThingLab, Beta, and Smalltalk, is motivated by the observation that ensuring internal consistency of objects is difficult when multiple state-changing methods are offered in the interface. Siri allowed objects to be defined through parameterized *constraint patterns*, and could establish maintain consistency between an object's fields given a declarative specification. This is similar to how we proposed to declare constraints during object initialization in Babelsberg (cf. Section 9.1).

Turtle [53] is a more recent CIP language written from scratch, while *Kaplan* [71] provides constraints in Scala. Both separate the declaration of *constrainable* variables from ordinary variables to make it clearer what may happen when a variable is used. Neither includes support for identity constraints, however. Like Babelsberg, the Turtle system provides constraint priorities; Kaplan does not. Because ordinary variables in Turtle are not determined by the solver, only constrainable variables have low-priority stay constraints on them. Kaplan does not currently support constraints over mutable types, so stay constraints are not relevant for it. Analogous to Kaleidoscope and in contrast to OCP, both languages separate constraint functions from ordinary functions. In Kaplan, only such specifically annotated functions can be used in constraints. Turtle does allow ordinary methods and variables in constraints; however, their values are treated as constants, making all ordinary methods work only in the forward direction. Kaplan does implement enumerating different solutions, a feature that we consider future work for the design presented in this work.

Rosette [117] is a solver-aided language that extends Racket with symbolic data types and solver operations on those types. Rosette integrates a solver for program synthesis as well as so-called "angelic execution." The latter is very similar to the work presented here, in that it attempts to find bindings for variables that are in an assertion to make that assertion true. As with Babelsberg, constraint expressions can contain arbitrary (imperative) functions that are partially evaluated in the host language to construct constraints. For the solving itself, both Rosette and Babelsberg handle object identity and mutable structures in a similar way. A more significant difference is that Rosette constraints are always asserted only at the time they are evaluated, unlike always in Babelsberg, and they cannot solve for and do not track re-assignment to ordinary variables, but only for separately declared "symbolic" variables.

SOUL [23] is a similar approach to the above, but more generally integrates logic programming and rule-based reasoning with a Smalltalk-like language. Logic predicates are written and can be used in message sends like Smalltalk methods, but rather than interpreting code imperatively, backtracking is used to change the binding of variables used in the rules. A major contribution of SOUL is its evaluation of a syntactic and semantic symbiosis between the two different language paradigms down to the level of method dispatch and variable declaration. Their observation that such a symbiosis is required to enable developers to utilize both paradigms freely when it makes sense motivated us to attempt a similar symbiosis in the work presented here.

Mozart/Oz [105] is a multi-paradigm language that supports functional, imperative, and logic programming. It also has a special focus on supporting concurrency, non-determinism, and search. In Oz, constraint programming is much more explicit than in an object-constraint language, with explicit variable types for unbound and finite-domain variables. Oz uses a constraint store that is monotonic, meaning that bindings and constraints can be added but not removed or changed, which is something we explicitly support in the work presented here. Another difference is that Oz explicitly allows non-determinism in constraint solving. Two features of Oz that we are missing in this work are explicit integration of threads of concurrent execution and constraint solving, and support for backtracking.

12.3. Constraint Programming with Libraries and DSLs

There is also a body of work that uses constraints in other ways in general-purpose programming languages, not as a linguistically integrated paradigm, but to provide new features through a DSL or library that integrates with an existing general purpose language.

Plan B [108] uses specifications as "reliable alternatives" to implementations, so that if an assertion fails, the system can use the specification as input to a constraint solver and continue execution. Similarly, *acceptability-oriented computing* (AOC) approaches use constraint solvers in their approach to correct a faulty program state automatically and continue running. (Thus, by replacing assertions with constraints, undesirable program states can be corrected by the runtime.) In both systems, constraint specifications were sometimes enough to ensure proper execution, and the actual imperative code in some methods could be partially or completely removed, leaving only the declarative specification of the correct result. A third system that uses constraints in this manner is *Squander* [89], where specifications of pre- and post-conditions around a method are handed to a solver, and if they are not met, the system state is updated with the help of the solver to meet them. In the Squander system, too, the authors propose that for some specification, the method body can actually be left empty and the specification of the intended effect of the method suffices. The use of constraints in these systems is mostly akin to the `once` constraint in our work, as a means to create a desired system state once, rather than continuously maintain it. An advantage of this approach is that the interactions of constraints with the imperative state are clearly localized.

BackTalk [106] and *Ilog Solver* (now IBM CPLEX) [63, 102] are other systems that aim to integrate a rich set of constraint solvers with imperative languages, but without aiming for full semantic integration. These systems provide a library for unified access to a wide range of solvers, and methods to combine these solvers with boolean operators or through propagation. Although constraints and imperative code are still written in the same syntax by making use of operator overloading and dynamic dispatch, they are clearly separated. The advantage of these systems is that the interface between the declarative and imperative world is explicit and all properties and features of the solvers are readily available.

For more specialized application domains, such as user interface definitions, systems like the *Auckland Layout Editor* (ALE) [82], *Mac OS X Auto Layout* [107] (which uses Cassowary to solve the constraints), or the Python GUI framework *Enaml* [27] provide separate DSLs that allow developers to express constraints on graphical objects and have the system maintain them. These DSLs hide the interaction between imperative changes to the GUI and the constraint solver. However, due to their specialized nature, this is not often a problem, and their clear focus on a limited domain has lead to the widespread use of such systems.

Finally, *αRby* [88] is a language that uses Ruby as a DSL for the Alloy specification language [67]. Its goal is reversed from the work presented here, in that it attempts to provide imperative constructs to Alloy users, that is, programmers familiar with constraints. It allows Alloy users to easily pre- or post-process their declarative models using imperative libraries, for example to experiment with visualizations. αRby translates Ruby programs into Alloy, but the programs are written in a style that closely mimics the Alloy language rather than with ordinary Ruby classes and methods. The mechanisms for abstraction and translation from Ruby are thus not relevant for Alloy users, and are not integrated with the Alloy language.

12.4. Dataflow Constraints and FRP

A number of contemporary application domains, including Web applications and graphical user interfaces, deal predominantly with interactive interfaces that have to react to and change in response to user input. Such behavior can be modeled with one-way data-flow constraints, and systems such as ThingLab and Siri already recognized this fact. Recent *Reactive Programming* [15, 13] systems try to address this need by focusing on data-dependencies between user events, interfaces, and the domain model, updating them when either changes. To that end, rather than integrating full constraint solvers in the way Babelsberg does, these languages allow programmers to subscribe to *event streams* (either continuous or discrete) and declare functional or data dependencies that are automatically maintained by the system.

Scratch [103], a tile-based scripting language, allows programmers to use "when" tiles to declare code that should run given a particular event stream currently has a value. For example, dragging can be implementations by moving a graphical object to the mouse position as long as the mouse button is down. *Lively Kernel/Webwerkstatt* [77], a Web-based, self-supporting development environment in the spirit of Self and Smalltalk, provides connections as a mechanism in JavaScript to manage data dependencies. Connections are used extensively in Lively to route user interaction events such as button presses or drags to trigger interface updates. Both these systems use one-way data-flow in a localized manner, triggering each event and reaction separately, and do not analyze, for example, the topology of connections (whereas a local propagation solver would do so to detect cyclic dependencies). This means that, in contrast to Babelsberg, these systems are more restrictive, but may also be easier to debug and understand internally.

Elm [19], *Fran* [26], and *Flapjax* [87] are recent examples of FRP systems. Flapjax is an extension to JavaScript that adds event streams as a kind of data source that can be connected to a function to update dependent values. Elm is a language created from scratch to facilitate user interfaces, and like Flapjax, is motivated heavily from JavaScript Web applications. Both these languages simplify the FRP theory by providing discrete event streams and allow events to be handled asynchronously. Fran, on the other hand, motivated in part by animations of physical processes, provides both continuous and discrete streams of values, and the systems takes care to sample continuous streams at the rate required to avoid missing important events (such as a bouncing ball hitting the floor). In these languages, the focus is on data dependencies, not desired system states as in Babelsberg. However, programmers can use constraint solvers to calculate downstream values in the functions triggered by a new value, so some support for constraint solving is available.

KScript [96] is a recent language syntactically modeled after JavaScript, that integrates FRP features with late-binding of variables and the Z3 constraint solver. In KScript, streams are connected to their dependencies by name, and name lookup happens any time a new value arrives from the stream, and the Z3 solver provides flexibility in calculating downstream values. KScript also deals with event cycles globally to allow cyclic dependencies, and thus provides capabilities to build graphical applications that are close to what we can achieve with Babelsberg, although using a different paradigm, rather than integrated with imperative code. The power to resolve variable names any time a new value arrives is not available in Babelsberg, and it would be interesting to see if such a feature could be used to make constraint expressions more re-usable.

12.5. Design Dimensions for Systems with Constraints and Related Mechanisms

While the paradigmatic perspective presented above is helpful in understanding the motivations behind systems that use constraints, it is useful to consider more basic design dimensions to compare languages directly rather than by their motivations and goals. We find that there are six dimensions that are useful to consider in this case. Table 12.1 presents an overview of the related work as well as the work presented here in terms of these design dimensions.

Constraint Building Blocks We first consider the basic building blocks for expressing constraints. All systems have some primitive relations (such as linear equations) that they support. A first question is whether these are expressed on special *symbolic variables* or directly on the *basic variables* of the underlying system. Second, is it possible to abstract over these primitive relations and provide *user-defined relations*. Finally, can constraint expressions include *functions*, *methods*, or *procedures* that are not designated constraint relations, and that are automatically transformed into a form suitable for the solver by the system.

Lowering When variables have constraints on them, there is a question if and how these can be used with functions in the system that do not use constraints. In reactive programming we talk about lifting with respect to the functions that work on ordinary and reactive variables. In this classification, we say that we lower constrained variables to concrete values. Lowering can be *implicit*, meaning that variables with constraints on them can be passed to ordinary functions and then exhibit one particular value that satisfies the current constraints. Alternatively, lowering can be *explicit*, meaning that the programmer has to get a concrete value from the constrained variable using some function before using it in non-declarative parts of the program.

Multi-directionality Most of the presented languages can solve at least some kinds of constraints multi-directionally. For some, this is true only for *primitive relations*, while others can also solve *user-defined relations* or even ordinary *functions*, *methods*, or *procedures* in multiple directions.

Solving Strategies Each constraint language needs at least one solving strategy, but many come with multiple solvers to allow solving more complex problems or to improve performance for specific types of constraints. The number of solvers is either *fixed* or *extensible* by the user. A related issue is whether, if there are multiple solvers, these can be used cooperatively (e.g., to solve constraints which are connected through shared variables) or if constraints in different solvers need to be satisfied separately.

Constraint Model Constraints may be viewed as *problems* to be solved or as relations to be maintained. Systems which do the latter react to some changes in the system by integrating the updated state with the active set of constraints. In that category there are systems that allow only those updates to the state that *refine* what is already known, leading to ever tighter bounds on the possible values of variables, whereas other systems react to *perturbations* of values by possibly changing some other state to keep the constraints satisfied or by rejecting the changed values.

An aspect to consider for the last group is how they deal with the frame problem, i.e., if and how parts of the system that do not need to change in response to constraint solving are kept at their current values. One option is to q ignore the problem and allow any perturbation of the system to cause all state to change in order to re-satisfy the constraints. A second option is to rely

on the memory model in that memory is unchanged if variables referring to it are not used during constraint solving and thus their memory is not written to. An explicit formalization is to use soft constraints on the entire system state to ask that all variables keep their current values unless a higher priority constraint contradicts this.

Answers or Solutions Under-constrained systems occur often in practice and a design decision is whether the system should produce *solutions* with concrete assignments for variables or if it should provide *answers*, i.e., results such as $10 \leq x \leq 20$ rather than a single value for x. And if there are multiple solutions, is the system capable of *enumerating* them or will it choose just one by some explicit or implicit mechanism.

⁂

We believe comparing the related approaches along these design dimensions highlights some of our initial goals in creating Babelsberg. Like the constraint-logic programming languages, Babelsberg integrates constraints with the underlying host language in such a way that we can use ordinary variables and methods both in our constraint expressions and in ordinary code. Compared to systems that use backtracking, however, we can only solve multi-directionally for methods that are well formed according to our design. The choice of not using backtracking to enable multi-directional solving in all cases is motivated by our desire to avoid surprising solutions as well as performance considerations. Third, while most languages only offer a fixed set of solvers, we think our cooperating solver has proven useful in practical applications to address a wider range of problems using constraints. Finally, akin to other constraint languages that deal with imperative state, Babelsberg has a perturbation model of constraints to connect changes to the system state automatically and deterministically with constraint solving and where the frame axioms are explicitly encoded using soft constraints.

Table 12.1.: A categorization of constraint systems

	Constraint Building Blocks	Lowering	Multi-directionality	Solving Strategies	Constraint Model	Answers or Solutions
αRby	symbolic variables, user-defined relations	manual	user-defined relations	fixed	solving	answers
ALE	primitive variables	implicit	primitive relations	fixed	perturbation[1]	solutions
AOC	symbolic variables	implicit	primitive relations	fixed	solving	solutions
Auto Layout	primitive variables	implicit	primitive relations	fixed	perturbation[1]	solutions
Babelsberg	variables, methods	implicit	all[5]	extensible, coop	perturbation[1]	solutions
Backtalk	symbolic variables, user-defined relations, methods	manual	all	extensible, coop	solving	enumerable solutions
CLP(X)	variables, procedures	implicit	all	extensible	solving	answers
Elm	behaviors, events, user-defined relations, functions	implicit	none	fixed	perturbation[2]	solutions
Enaml	primitive variables	implicit	primitive relations	fixed	perturbation[1]	solutions
Flapjax	behaviors, events, user-defined relations, functions	implicit	none	fixed	perturbation[2]	solutions
Fran	behaviors, events, user-defined relations, functions	explicit	none	fixed	perturbation[3]	solutions
Ilog (CPLEX)	symbolic variables, user-defined relations, methods	manual	user-defined relations	fixed	solving	enumerable solutions
KScript	behaviors, events, user-defined relations, functions	implicit	none	fixed	perturbation[2]	solutions
Kaleidoscope	variables, user-defined relations	implicit	user-defined relations	fixed, coop	perturbation[1]	solutions
Kaplan	symbolic variables, user-defined relations, methods	manual	user-defined relations	fixed	perturbation[4,6]	solutions
Lively	variables, methods	implicit	none	fixed	perturbation[2]	solutions
Mozart/Oz	variables, user-defined relations	implicit	user-defined relations	extensible, coop	refinement	answers
Plan B	symbolic variables, user-defined relations	implicit	user-defined relations	fixed	solving	solutions
Rosette	symbolic variables, user-defined relations, functions	explicit	user-defined relations	fixed	solving	solutions
Scratch	events, functions	implicit	none	fixed	perturbation[2]	solutions
Siri	variables, user-defined relations	implicit	user-defined relations	fixed	perturbation[2]	solutions
Sketchpad	primitive variables	implicit	primitive relations	fixed, coop	perturbation[2]	solutions
Squander	symbolic variables, user-defined relations	implicit	user-defined relations	fixed	solving	enumerable solutions
ThingLab	variables, user-defined relations	implicit	user -defined relations	fixed, coop	perturbation[2]	solutions
Turtle	symbolic variables, user-defined relations, methods	explicit	user-defined relations	fixed, coop	perturbation[1,6]	solutions

[1] Soft constraints as frame axioms
[2] Implicit reliance on memory model for frame axioms
[3] Constraints over time as frame axioms
[4] No frame axioms
[5] Multi-directional solving through imperative methods is limited as per our design (cf. Section 4.2)
[6] Perturbation only of symbolic variables, ordinary variables are frozen by-value into the constraints

13. Summary and Outlook

13.1. 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 themselves, both small and large. Among the smaller issues left for future work are questions around debugging, comprehension, and the choice of solvers.

We mentioned debugging as a long-standing problem in constraint programming, and although we have presented prototypes of inspectors and debuggers, our system, too, suffers from poor debugging support. Our tools treat the solver as a black box but show which objects are affected by a given set of constraints, which constraints are relevant in a given scope, and what value changes trigger constraint satisfaction. While this can help in understanding *when* and *how* the solver affects the system, it does not explain *why* it selected a particular solution. exploring options to visualize the solving process itself, to step through the decisions of the solver, and to allow the programmer to explore alternative solutions is left for future work.

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, maybe as a Lint-style tool for Babelsberg languages. We already provide special error messages for a few operations that 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 $\leq$, so our system mentions this in the error message. Future work could extend how these error messages are generated to offer additional alternatives and guidance, possibly using ontologies of available constraint solvers and thus teaching the programmer about their capabilities and limits.

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 have presented a number of schemes to select from the available solvers. More research is required to find better ways to automatically choose the right combination of solvers based on a variety of metrics besides those centered around performance and quality that we have presented. Which metrics actually make sense to use are not clear as of this point and might vary based on the needs of the developer or the application, suggesting that an interactive dialog between the development environment and the developer might be required to choose the right solver.

A larger question that this work leaves unanswered is how to integrate Prolog-style backtracking into the constraint solving process. Right now, constraint satisfaction leads to exactly one solution which is then visible to the imperative parts of the program. An option to support finding multiple solutions would be to design an explicit new mechanism for querying and choosing between multiple solutions, possibly akin to the Mozart/Oz system, where multiple threads of execution can be spawned from multiple solutions. An alternative is to instead use Prolog as the basis, and interact with the Prolog runtime for dynamically adding and removing constraints to find multiple

solutions. The main program will still be an object constraint program, while the Prolog part is a separate sub-component, with some clearly defined means to communicate with the main program. *Unipycation* [5], which is a composition of Python and Prolog, seems like a good starting point for such future work, since that work already looked at many issues about how a dynamically-typed imperative language and Prolog can interact.

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, to aid, for example, the programming of self-healing systems.

13.2. Conclusions

We have presented Babelsberg, a design for a family of Object-Constraint Programming languages that extends existing object-oriented languages to support constraints. Our design unifies the language constructs for constraint definition and object-oriented code, it provides automatic maintenance of constraints, integrates with the existing syntax and semantics of the object-oriented host language, an uses a communication interface with the solver that makes it easy to add new solvers and constraint solver constructs such as read-only variables and incremental re-solving.

As part of our design, we have presented design principles to control the power of the solver in object-constraint programming languages to avoid surprising or non-deterministic behavior. Such behavior had often been a problem in prior work that attempted to integrate constraint solving with imperative languages. Our design principles ensure that object structure and identity are preserved when adding constraints, and that any changes to them are deterministic. Our principles also include a small set of informal rules that guide developers in expressing constraints, in particular those involving message sends and object identity comparisons.

Finally, our design also includes support for cooperating constraint solvers to address a wider variety of application domains than any one solver can. The architecture for cooperating solvers that we use has only minimal requirements on the solvers and thus supports flexible combinations of existing solvers, as well as user-defined, highly specialized, and domain-specific solvers in its framework. Keeping with our goal of making programming with constraints accessible to imperative programmers, our cooperating solvers architecture includes designs for automatically selecting solvers for given problems based on different heuristics, and combining these solvers

We have formalized core principles of our design in a natural semantics that defines a useful subset of a full object-constraint language and is meant to highlight key properties to inform practical implementations. Its description includes formal rules for defining constraints on the results of message sends, constraints involving structure and identity, integration with control flow, as well as constraints over collections. The restrictions are easy to understand, but may, in practice, be too conservative, as experience with writing Babelsberg applications has shown that it might

be useful to relax some rules for concrete implementations or application domains. Thus, our recommendation is to use the formalism as a guide for language implementers but allow deviations from it if the guarantees of determinism can also be relaxed. We also presented an executable semantics that can run a suite of OCP example programs and that provides a mechanism to generate a language test suite to check practical implementations for conformance.

As a practical evaluation for our design, we have presented two implementation approaches for Babelsberg and discussed key implementation details that serve to provide practical performance without requiring intimate developer knowledge of constraint solver performance or implementation details. The differentiating property of our language implementations is whether they are based on a custom VM or are done as a library. The main advantage of the library-based approach seems to be its applicability to existing runtimes, and that it thus integrates well with existing applications. A further important advantage may be that the meta-constraint API is entirely implemented in the language, and it is thus straightforward to create and develop tools for it from within the language itself.

However, our experience with library-based implementations is limited by the fact that we have based our Babelsberg libraries on JavaScript and Squeak/Smalltalk, both dynamic, object-oriented languages languages. These languages are dynamic enough to allow three specific features, which we rely on in our implementations. First, both languages support (through property accessors or method wrappers) a means to intercept (field) variable lookup, so access to constrained variables can be redirected to solver objects. Compared to the VM-based approach, the Babelsberg libraries are more limited, since we cannot put constraints on local variables, only on fields. Furthermore, with access to the VM we can optimize the storage for solver objects to work well with a specific JIT. When running on different VMs, however, we need to rely on more general optimizations. Second, a VM-based implementation of Babelsberg could opt for a simpler implementation and ignore encapsulation to modify data structures directly. However, a Babelsberg library must use a more complicated implementation. For Smalltalk and JavaScript, access to fields of ordinary mutable objects is possible, but for opaque VM objects (such as files or sockets), our library-based implementations have to call the appropriate API functions to manipulate the data structures. Third, the host language must provide a means to modify interpretation of a block of code to implement the constraint construction mode. This is most easily done in languages that provide closures, or where the language provides reflective access to the bytecode of a method. In precompiled languages, however, it is not clear if this is possible.

Finally, we presented applications and development experience with building Babelsberg applications, including tentative ideas for programming patterns and prototypes of development tools. Our experience indicates that Babelsberg is indeed useful and can make code clearer and shorter in a number of application domains. Yet, from our limited experience with building Babelsberg applications, we cannot give a clear recommendation as to which implementation strategy is more suitable for OCP languages. Both presented approaches have advantages, and their limitations seem not to be strong disadvantages. (The latter actually makes the implementation of inlining collection predicates easier, because the loop counter is just used as a constant in each iteration.)

In summary, we argue that this works' approach at providing constraints integrated with an object-oriented language has proven useful for a number of applications and can be a useful incremental addition to current programming practice.

Publications

Journals

- Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. "Constraining Timing-dependent Communication for Debugging Non-deterministic Failures". In: *Science of Computer Programming (SCICO)* 11.6 (2015). DOI: 10.1016/j.scico.2015.11.006
- Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. "Visibility of Context-oriented Behavior and State in L". In: *Computer Software JSSST Journal* 32.3 (2015), pp. 149–159. DOI: 10.11309/jssst.32.3_149
- Tim Felgentreff, Alan Borning, and Robert Hirschfeld. "Specifying and Solving Constraints on Object Behavior". In: *Journal of Object Technology (JOT)* 13.4 (2014), pp. 1–38. DOI: 10.5381/jot.2014.13.4.a1

Conferences

- Tim Felgentreff, Todd D. Millstein, Alan Borning, and Robert Hirschfeld. "Checks and Balances: Constraint Solving Without Surprises in Object-Constraint Programming Languages". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2015, pp. 767–782. DOI: 10.1145/2814270.2814311
- Tim Felgentreff, Tobias Pape, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. "Language Independent Storage Strategies for Tracing JIT based VMs". In: *Dynamic Languages Symposium (DLS)*. ACM, 2015, pp. 119–128. DOI: 10.1145/2816707.2816716
- Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. "Visibility of Context-oriented Behavior and State in L". In: *Annual Conference of the Japan Society for Software Science and Technology (JSSST)*. 2014
- Bastian Steinert, Lauritz Thamsen, Tim Felgentreff, and Robert Hirschfeld. "Object Versioning to Support Recovery Needs: Using Proxies to Preserve Previous Development States in Lively". In: *Dynamic Languages Symposium (DLS)*. ACM, 2014, pp. 113–124. DOI: 10.1145/2661088.2661093
- Bert Freudenberg, Daniel H. H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser". In: *Dynamic Languages Symposium (DLS)*. ACM, 2014, pp. 57–66. DOI: 10.1145/2661088.2661100
- Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. "Babelsberg/JS — A Browser-Based Implementation of an Object Constraint Language". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2014, pp. 411–436. DOI: 10.1007/978-3-662-44202-9_17
- Michael Perscheid, Tim Felgentreff, and Robert Hirschfeld. "Follow The Path: Debugging State Anomalies Along Execution Histories". In: *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 124–133. DOI: 10.1109/csmr-wcre.2014.6747162

Workshops

- Tim Felgentreff, Stefan Lehmann, Robert Hirschfeld, Sebastian Gerstenberg, Jakob Reschke, Lars Rückert, Patrick Siegler, Jan Graichen, Christian Nicolai, and Malte Swart. "Automatically Selecting and Optimizing Constraint Solver Procedures for Object-Constraint Languages". In: *Constrained and Reactive Objects Workshop (CROW)*. ACM, 2016. DOI: 10.1145/2892664.2892671
- Stefan Lehmann, Tim Felgentreff, Jens Lincke, Patrick Rein, and Robert Hirschfeld. "Reactive Object Queries". In: *Constrained and Reactive Objects Workshop (CROW)*. ACM, 2016. DOI: 10.1145/2892664.2892665
- Bastian Kruck, Stefan Lehmann, Christoph Kessler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. "Multi-level Debugging for Interpreter Developers". In: *Workshop on Language Modularity A La Mode (LaMOD)*. ACM, 2016. DOI: 10.1145/2892664.2892679
- Tim Felgentreff, Jens Lincke, Robert Hirschfeld, and Lauritz Thamsen. "Lively Groups: Shared Behavior in a World of Objects Without Classes or Prototypes". In: *Future Programming Workshop (FPW)*. ACM, 2015. DOI: 10.1145/2846656.2846659
- Stefan Lehmann, Tim Felgentreff, and Robert Hirschfeld. "Connecting Object Constraints with Context-oriented Programming: Scoping Constraints with Layers and Activating Layers with Constraints". In: *International Workshop on Context-Oriented Programming (COP)*. ACM, 2015, 1:1–1:6. DOI: 10.1145/2786545.2786549
- Fabio Niephaus, Matthias Springer, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Call-target-specific Method Arguments". In: *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM, 2015
- Tim Felgentreff, Tobias Pape, Lars Wassermann, Robert Hirschfeld, and Carl Friedrich Bolz. "Towards Reducing the Need for Algorithmic Primitives in Dynamic Language VMs Through a Tracing JIT". In: *Workshop on Implementation, Compilation, and Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM, 2015. DOI: 10.1145/2843915.2843924
- Maria Graber, Tim Felgentreff, Robert Hirschfeld, and Alan Borning. "Solving Interactive Logic Puzzles With Object-Constraints — An Experience Report Using Babelsberg/S for Squeak/Smalltalk". In: *Workshop on Reactive and Event-based Languages & Systems (REBLS)*. 2014, 1:1–1:5
- Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld. "Applying Data-driven Tool Development to Context-oriented Languages". In: *International Workshop on Context-Oriented Programming (COP)*. ACM, 2014, 1:1–1:7. DOI: 10.1145/2637066.2637067
- Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. "Constraining Timing-dependent Communication for Debugging Non-deterministic Failures". In: *Workshop on Academic Software Development Tools and Techniques (WASDeTT)*. 2013

Technical Reports

- Tim Felgentreff, Robert Hirschfeld, Alan Borning, and Millstein Todd. *Babelsberg/RML: Executable Semantics and Language Testing with RML*. Tech. rep. 103. Hasso Plattner Institute, 2015
- Tim Felgentreff. "Checks and Balances: Object-Constraints Without Surprises". In: *Proceedings of the 9th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 100. Hasso Plattner Institute, 2016

- Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Robert Hirschfeld, and Carsten Witt. *ecoControl: Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern*. Tech. rep. 93. Hasso Plattner Institute, 2015
- Tim Felgentreff, Todd Millstein, and Alan Borning. *Developing a Formal Semantics for Babelsberg: A Step-by-Step Approach*. Tech. rep. 2015-002b. Viewpoints Research Institute, 2014
- Tim Felgentreff. "Implementing an Object-Constraint Extension Without VM Support". In: *Proceedings of the 8th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 95. Hasso Plattner Institute, 2015
- Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: Specifying and Solving Constraints on Object Behavior*. Tech. rep. 2013-001. Viewpoints Research Institute, 2013
- Tim Felgentreff. "Specifying Multi-domain Constraints on Object-Behavior". In: *Proceedings of the 7th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 83. Hasso Plattner Institute, 2014

Bibliography

[1] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. "Event-Specific Software Composition in Context-Oriented Programming". In: *International Conference on Software Composition (SC)*. Springer, 2010, pp. 50–65. DOI: 10.1007/978-3-642-14046-4_4.

[2] Edward A. Ashcroft and William W. Wadge. "Lucid, a Nonprocedural Language with Iteration". In: *Communications of the ACM* 20.7 (1977), pp. 519–526. DOI: 10.1145/359636.359715.

[3] Greg J. Badros, Alan Borning, and Peter J. Stuckey. "The Cassowary Linear Arithmetic Constraint Solving Algorithm". In: *ACM Transactions on Computer-Human Interaction* 8.4 (2001), pp. 267–306. DOI: 10.1145/504704.504705.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Department of Computer Science, The University of Iowa, 2015.

[5] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. "Unipycation: A Case Study in Cross-language Tracing". In: *Workshop on Virtual Machines and Intermediate Languages*. ACM, 2013, pp. 31–40. DOI: 10.1145/2542142.2542146.

[6] Nikolaj Björner and Anh-Dung Phan. "*νZ*–Maximal Satisfaction with Z3". In: *International Symposium on Symbolic Computation in Software Science (SCSS)*. 2014.

[7] Alan Borning. *Architectures for Cooperating Constraint Solvers*. Tech. rep. M-2012-003. Viewpoints Research Institute, 2012.

[8] Alan Borning. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". In: *ACM Transactions on Programming Languages and Systems* 3.4 (1981), pp. 353–387. DOI: 10.1145/357146.357147.

[9] Alan Borning and Greg Badros. "On Finding Graphically Plausible Solutions to Constraint Hierarchies: The Split Stay Problem". In: *Workshop on Soft Constraints: Theory and Practice*. 2000.

[10] Alan Borning and Bjørn N. Freeman-Benson. "Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics". In: *Constraints* 3.1 (1998), pp. 9–32. DOI: 10.1023/a:1009704614502.

[11] Alan Borning, Bjørn N. Freeman-Benson, and Molly Wilson. "Constraint Hierarchies". In: *Lisp and Symbolic Computation* 5.3 (1992), pp. 223–270. DOI: 10.1007/bf01807506.

[12] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. *Cascading Style Sheets, level 2*. W3C Working Draft. http://www.w3.org/TR/WD-css2/. 1998.

[13] Frédéric Boussinot, Guillaume Doumenc, and Jean-Bernard Stefani. "Reactive Objects". In: *Annales des télécommunications*. 1996, pp. 459–473.

[14] Michael Carbin and Martin C. Rinard. "Automatically Identifying Critical Input Regions and Code in Applications". In: *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2010, pp. 37–48. DOI: 10.1145/1831708.1831713.

[15] Denis Caromel and Yves Roudier. "Reactive Programming in Eiffel//". In: *France-Japan Workshop on Object-Based Parallel and Distributed Computation (OBPDC)*. Springer, 1995, pp. 125–147. DOI: 10.1007/3-540-61487-7_25.

[16] Craig Chambers, David Ungar, and Elgin Lee. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes". In: *Lisp and Symbolic Computation* 4.3 (1991), pp. 243–281. DOI: 10.1145/74877.74884.

[17] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. "Satisfiability Modulo the Theory of Costs: Foundations and Applications". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2010, pp. 99–113. DOI: 10.1007/978-3-642-12002-2_8.

[18] Pascal Costanza and Robert Hirschfeld. "Language Constructs for Context-oriented Programming: An Overview of ContextL". In: *Dynamic Languages Symposium (DLS)*. ACM, 2005, pp. 1–10. DOI: 10.1145/1146841.1146842.

[19] Evan Czaplicki and Stephen Chong. "Asynchronous Functional Reactive Programming for GUIs". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 411–422. DOI: 10.1145/2462156.2462161.

[20] Martin Davis, George Logemann, and Donald W. Loveland. "A Machine Program for Theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397. DOI: 10.1145/368273.368557.

[21] Brian Demsky and Martin C. Rinard. "Goal-Directed Reasoning for Specification-Based Data Structure Repair". In: *IEEE Transactions on Software Engineering* 32.12 (2006), pp. 931–951. DOI: 10.1109/tse.2006.122.

[22] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog - The Standard: Reference Manual*. Springer, 1996. ISBN: 978-3-540-59304-1.

[23] Maja D'Hondt, Kris Gybels, and Viviane Jonckers. "Seamless Integration of Rule-based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis". In: *Symposium on Applied Computing (SAC)*. ACM, 2004, pp. 1328–1335. DOI: 10.1145/967900.968168.

[24] Anne Dinning and Edith Schonberg. "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection". In: *Symposium on Princiles & Practice of Parallel Programming (PPOPP)*. ACM, 1990, pp. 1–10. DOI: 10.1145/99163.99165.

[25] Henry Dudeney. "Send More Money". In: *Strand Magazine* 214 (1924), pp. 68–97.

[26] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming (ICFP)*. ACM, 1997, pp. 263–273. DOI: 10.1145/258948.258973.

[27] Enthought Inc. *Enaml 0.6.3 Documentation*. 2014. URL: http://docs.enthought.com/enaml/.

[28] Tim Felgentreff. "Checks and Balances: Object-Constraints Without Surprises". In: *Proceedings of the 9th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 100. Hasso Plattner Institute, 2016.

[29] Tim Felgentreff. "Implementing an Object-Constraint Extension Without VM Support". In: *Proceedings of the 8th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 95. Hasso Plattner Institute, 2015.

[30] Tim Felgentreff. "Specifying Multi-domain Constraints on Object-Behavior". In: *Proceedings of the 7th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*. Tech. rep. 83. Hasso Plattner Institute, 2014.

[31] Tim Felgentreff. *Topaz Ruby*. `http://lanyrd.com/2013/wrocloverb/sccygw/`. Invited Talk at Wroclove.rb. 2013.

[32] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. *Babelsberg: Specifying and Solving Constraints on Object Behavior*. Tech. rep. 2013-001. Viewpoints Research Institute, 2013.

[33] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. "Specifying and Solving Constraints on Object Behavior". In: *Journal of Object Technology (JOT)* 13.4 (2014), pp. 1–38. DOI: 10.5381/jot.2014.13.4.a1.

[34] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. "Babelsberg/JS — A Browser-Based Implementation of an Object Constraint Language". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2014, pp. 411–436. DOI: 10.1007/978-3-662-44202-9_17.

[35] Tim Felgentreff, Robert Hirschfeld, Alan Borning, and Millstein Todd. *Babelsberg/RML: Executable Semantics and Language Testing with RML*. Tech. rep. 103. Hasso Plattner Institute, 2015.

[36] Tim Felgentreff, Stefan Lehmann, Robert Hirschfeld, Sebastian Gerstenberg, Jakob Reschke, Lars Rückert, Patrick Siegler, Jan Graichen, Christian Nicolai, and Malte Swart. "Automatically Selecting and Optimizing Constraint Solver Procedures for Object-Constraint Languages". In: *Constrained and Reactive Objects Workshop (CROW)*. ACM, 2016. DOI: 10.1145/2892664.2892671.

[37] Tim Felgentreff, Jens Lincke, Robert Hirschfeld, and Lauritz Thamsen. "Lively Groups: Shared Behavior in a World of Objects Without Classes or Prototypes". In: *Future Programming Workshop (FPW)*. ACM, 2015. DOI: 10.1145/2846656.2846659.

[38] Tim Felgentreff, Todd D. Millstein, Alan Borning, and Robert Hirschfeld. "Checks and Balances: Constraint Solving Without Surprises in Object-Constraint Programming Languages". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2015, pp. 767–782. DOI: 10.1145/2814270.2814311.

[39] Tim Felgentreff, Todd Millstein, and Alan Borning. *Developing a Formal Semantics for Babelsberg: A Step-by-Step Approach*. Tech. rep. 2015-002b. Viewpoints Research Institute, 2014.

[40] Tim Felgentreff, Todd Millstein, Alan Borning, and Robert Hirschfeld. *Checks and Balances — Constraint Solving Without Surprises in Object-Constraint Programming Languages: Full Formal Development*. Tech. rep. 2015-001. Viewpoints Research Institute, 2015.

[41] Tim Felgentreff, Tobias Pape, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. "Language Independent Storage Strategies for Tracing JIT based VMs". In: *Dynamic Languages Symposium (DLS)*. ACM, 2015, pp. 119–128. DOI: 10.1145/2816707.2816716.

[42] Tim Felgentreff, Tobias Pape, Lars Wassermann, Robert Hirschfeld, and Carl Friedrich Bolz. "Towards Reducing the Need for Algorithmic Primitives in Dynamic Language VMs Through a Tracing JIT". In: *Workshop on Implementation, Compilation, and Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM, 2015. DOI: 10.1145/2843915.2843924.

[43] Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. "Constraining Timing-dependent Communication for Debugging Non-deterministic Failures". In: *Workshop on Academic Software Development Tools and Techniques (WASDeTT)*. 2013.

[44] Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. "Constraining Timing-dependent Communication for Debugging Non-deterministic Failures". In: *Science of Computer Programming (SCICO)* 11.6 (2015). DOI: 10.1016/j.scico.2015.11.006.

[45] Bjørn N. Freeman-Benson. "Kaleidoscope: Mixing Objects, Constraints and Imperative Programming". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 1990, pp. 77–88. DOI: 10.1145/97945.97957.

[46] Bjørn N. Freeman-Benson and Alan Borning. "Integrating Constraints with an Object-Oriented Language". In: *European Conference on Object-Oriented Programming (ECOOP)*. 1992, pp. 268–286. DOI: 10.1007/bfb0053042.

[47] Bjørn N. Freeman-Benson and Alan Borning. "The Design and Implementation of Kaleidoscope'90-A Constraint Imperative Programming Language". In: *International Conference on Computer Languages (ICCL)*. IEEE, 1992, pp. 174–180. DOI: 10.1109/iccl.1992.185480.

[48] Bjorn Freeman-Benson and John Maloney. "The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver". In: *Annual Phoenix Conference on Computers and Communications*. IEEE, 1989, pp. 538–542. DOI: 10.1109/pccc.1989.37442.

[49] Bert Freudenberg, Daniel H. H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser". In: *Dynamic Languages Symposium (DLS)*. ACM, 2014, pp. 57–66. DOI: 10.1145/2661088.2661100.

[50] Eugene C. Freuder. "In Pursuit of the Holy Grail". In: *Constraints* 2.1 (1997), pp. 57–61. DOI: 10.1023/a:1009749006768.

[51] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. "Trace-based Just-in-time Type Specialization for Dynamic Languages". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, pp. 465–478. DOI: 10.1145/1542476.1542528.

[52] Maria Graber, Tim Felgentreff, Robert Hirschfeld, and Alan Borning. "Solving Interactive Logic Puzzles With Object-Constraints — An Experience Report Using Babelsberg/S for Squeak/Smalltalk". In: *Workshop on Reactive and Event-based Languages & Systems (REBLS)*. 2014, 1:1–1:5.

[53] Martin Grabmüller and Petra Hofstedt. "Turtle: A Constraint Imperative Programming Language". In: *Research and Development in Intelligent Systems (RDIS)*. Springer, 2004, pp. 185–198. DOI: 10.1007/978-0-85729-412-8_14.

[54] Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Robert Hirschfeld, and Carsten Witt. *ecoControl: Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern*. Tech. rep. 93. Hasso Plattner Institute, 2015.

[55] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context-oriented Programming". In: *Journal of Object Technology (JOT)* 7.3 (2008), pp. 125–151. DOI: 10.5381/jot.2008.7.3.a4.

[56] Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. "Visibility of Context-oriented Behavior and State in L". In: *Annual Conference of the Japan Society for Software Science and Technology (JSSST)*. 2014.

[57] Robert Hirschfeld, Hidehiko Masuhara, Atsushi Igarashi, and Tim Felgentreff. "Visibility of Context-oriented Behavior and State in L". In: *Computer Software JSSST Journal* 32.3 (2015), pp. 149–159. DOI: 10.11309/jssst.32.3_149.

[58] Bruce Horn. "Constraint Patterns As a Basis for Object-Oriented Programming". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 1992, pp. 218–233. DOI: 10.1145/141936.141955.

[59] Bruce Horn. "Properties of User Interface Systems and the Siri Programming Language". In: *Languages for Developing User Interfaces*. Jones and Bartlett, 1992, pp. 211–236.

[60] Hiroshi Hosobe. "A Modular Geometric Constraint Solver for User Interface Applications". In: *Annual Symposium on User Interface Software and Technology (UIST)*. ACM, 2001, pp. 91–100. DOI: 10.1145/502348.502362.

[61] Hiroshi Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. "Generalized Local Propagation: A Framework for Solving Constraint Hierarchies". In: *International Conference on Principles and Practice of Constraint Programming (PPCP)*. Springer, 1996, pp. 237–251. DOI: 10.1007/3-540-61551-2_78.

[62] Scott E. Hudson and Ian E. Smith. "Ultra-Lightweight Constraints". In: *Annual Symposium on User Interface Software and Technology (UIST)*. ACM, 1996, pp. 147–155. DOI: 10.1145/237091.237112.

[63] *ILOG CPLEX Optimization Studio*. IBM. 2014.

[64] Daniel H. H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 1978, pp. 9–16. DOI: 10.1145/512760.512762.

[65] Daniel H. H. Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 1997, pp. 318–326. DOI: 10.1145/263698.263754.

[66] Daniel H. H. Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The Lively Kernel: A Self-supporting System on a Web Page". In: *Workshop on Self-Sustaining Systems (S3)*. Springer, 2008, pp. 31–50. DOI: 10.1007/978-3-540-89275-5_2.

[67] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Transactions on Software Engineering Methodologies* 11.2 (2002), pp. 256–290. DOI: 10.1145/505145.505149.

[68] Joxan Jaffar and Jean-Louis Lassez. "Constraint Logic Programming". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 1987, pp. 111–119. DOI: 10.1145/41625.41635.

[69] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. "The CLP($\mathcal{R}$) Language and System". In: *ACM Transactions on Programming Languages and Systems* 14.3 (1992), pp. 339–395. DOI: 10.1145/129393.129398.

[70] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. "EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition". In: *International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 253–264. DOI: 10.1145/1960275.1960305.

[71] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. "Constraints as Control". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2012, pp. 151–164. DOI: 10.1145/2103656.2103675.

[72] Bastian Kruck, Stefan Lehmann, Christoph Kessler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. "Multi-level Debugging for Interpreter Developers". In: *Workshop on Language Modularity A La Mode (LaMOD)*. ACM, 2016. DOI: 10.1145/2892664.2892679.

[73] Stefan Lehmann, Tim Felgentreff, and Robert Hirschfeld. "Connecting Object Constraints with Context-oriented Programming: Scoping Constraints with Layers and Activating Layers with Constraints". In: *International Workshop on Context-Oriented Programming (COP)*. ACM, 2015, 1:1–1:6. DOI: 10.1145/2786545.2786549.

[74] Stefan Lehmann, Tim Felgentreff, Jens Lincke, Patrick Rein, and Robert Hirschfeld. "Reactive Object Queries". In: *Constrained and Reactive Objects Workshop (CROW)*. ACM, 2016. DOI: 10.1145/2892664.2892665.

[75] Jens Lincke. "Evolving Tools in a Collaborative Self-supporting Development Environment". Submitted. PhD thesis. Hasso Plattner Institut, Universität Potsdam, 2014.

[76] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. "An Open Implementation for Context-oriented Layer Composition in ContextJS". In: *Science of Computer Programming (SCICO)* 76.12 (2011), pp. 1194–1209. DOI: 10.1016/j.scico.2010.11.013.

[77] Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, Marko Röder, and Robert Hirschfeld. "The Lively PartsBin: A Cloud-Based Repository for Collaborative Development of Active Web Content". In: *Hawaii International International Conference on Systems Science (HICSS)*. IEEE, 2012, pp. 693–701. DOI: 10.1109/hicss.2012.42.

[78] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin C. Rinard. "Automatic Input Rectification". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 80–90. DOI: 10.1109/icse.2012.6227204.

[79] Gus Lopez, Bjørn N. Freeman-Benson, and Alan Borning. "Constraints and Object Identity". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1994, pp. 260–279. DOI: 10.1007/bfb0052187.

[80] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. "Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 1994, pp. 259–271. DOI: 10.1145/191081.191118.

[81] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. "Context-oriented Programming: Beyond Layers". In: *International Conference on Dynamic Languages (ICDL)*. ACM, 2007, pp. 143–156. DOI: 10.1145/1352678.1352688.

[82] Christof Lutteroth and Gerald Weber. "End-user GUI customization". In: *International Conference on Computer-Human Interaction (CHI)*. ACM, 2008, pp. 1–8. DOI: 10.1145/1496976.1496977.

[83] Inês Lynce and Joël Ouaknine. "Sudoku as a SAT Problem". In: *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*. Springer, 2006, pp. 1–9. DOI: 10.1.1.331.458.

[84] Michael J. Maher. "Logic Semantics for a Class of Committed-Choice Programs". In: *International Conference on Logic Programming*. 1987, pp. 858–876.

[85] John Maloney. *Morphic: The Self user interface framework*. 4th. Self. 1995.

[86] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* 2.4 (1976), pp. 308–320. DOI: 10.1109/tse.1976.233837.

[87] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. "Flapjax: A Programming Language for Ajax Applications". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2009, pp. 1–20. DOI: 10.1145/1640089.1640091.

[88] Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. "αRby–An Embedding of Alloy in Ruby". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 56–71.

[89] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. "Unifying Execution of Imperative and Declarative Code". In: *International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 511–520. DOI: 10.1145/1985793.1985863.

[90] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[91] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad T. Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces". In: *IEEE Computer* 23.11 (1990), pp. 71–85. DOI: 10.1109/2.60882.

[92] Bernard A Nadel. "Tree Search and Arc Consistency in Constraint Satisfaction Algorithms". In: *Search in Artificial Intelligence*. Springer, 1988, pp. 287–342.

[93] Greg Nelson and Derek C. Oppen. "Simplification by Cooperating Decision Procedures". In: *ACM Transactions on Programming Languages and Systems* 1.2 (1979), pp. 245–257. DOI: 10.1145/357073.357079.

[94] Fabio Niephaus, Matthias Springer, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Call-target-specific Method Arguments". In: *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM, 2015.

[95] Robert Nieuwenhuis and Albert Oliveras. "On SAT Modulo Theories and Optimization Problems". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2006, pp. 156–169. DOI: 10.1007/11814948_18.

[96] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. "KScript and KSWorld: A Time-aware and Mostly Declarative Language and Interactive GUI Framework". In: *Symposium on New Ideas in Programming and Reflections on Software (Onward!)* ACM, 2013, pp. 117–134. DOI: 10.1145/2509578.2509590.

[97] CWAM van Overveld. "30 Years after Sketchpad: Relaxation of Geometric Constraints Revisited". In: *CWI Quarterly* 6.4 (1993), pp. 363–383.

[98] Francois Pachet and Pierre Roy. "Integrating Constraint Satisfaction Techniques with Complex Object Structures". In: *Annual Conference of the British Computer Society Specialist Group on Expert Systems*. Cambridge University Press, 1995, pp. 11–22.

[99] Michael Perscheid, Tim Felgentreff, and Robert Hirschfeld. "Follow The Path: Debugging State Anomalies Along Execution Histories". In: *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 124–133. DOI: 10.1109/csmr-wcre.2014.6747162.

[100] Mikael Pettersson. "RML — A New Language and Implementation for Natural Semantics". In: *International Symposium on Programming Language Implementation and Logic Programming (PLILP)*. 1994, pp. 117–131. DOI: 10.1007/3-540-58402-1_10.

[101] Patrick Prosser. "Hybrid Algorithms for the Constraint Satisfaction Problem". In: *Computational Intelligence* 9.3 (1993), pp. 268–299. DOI: 10.1111/j.1467-8640.1993.tb00310.x.

[102] Jean-François Puget. *A C++ Implementation of CLP*. Tech. rep. ILOG, 1994. DOI: 10.1.1.15.9273.

[103] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. "Scratch: Programming for All". In: *Communications of the ACM* 52.11 (2009), pp. 60–67. DOI: 10.1145/1592761.1592779.

[104] Armin Rigo and Samuele Pedroni. "PyPy's Approach to Virtual Machine Construction". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006, pp. 944–953. DOI: 10.1145/1176617.1176753.

[105] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. "Logic Programming in the Context of Multiparadigm Programming: The Oz Experience". In: *Theory and Practice of Logic Programming (TPLP)* 3.6 (2003), pp. 715–763. DOI: 10.1017/s1471068403001741.

[106] Pierre Roy and François Pachet. "Reifying Constraint Satisfaction in Smalltalk". In: *Journal of Object-oriented Programming (JOOP)* 10.4 (1997), pp. 43–51, 63.

[107] Erica Sadun. *iOS Auto Layout Demystified*. 1st ed. Addison-Wesley, 2013. ISBN: 9780321967190.

[108] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. "Falling Back on Executable Specifications". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2010, pp. 552–576. DOI: 10.1007/978-3-642-14107-2_26.

[109] Michael Sannella. "The SkyBlue Constraint Solver and Its Applications". In: *International Conference on Principles and Practice of Constraint Programming (PPCP)*. University of Washington, 1993, pp. 258–268.

[110] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993. ISBN: 0897913434. DOI: 10.1145/96709.96733.

[111] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. "Semantic Foundations of Concurrent Constraint Programming". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 1991, pp. 333–352. DOI: 10.1145/99583.99627.

[112] Chris Seaton, Michael L. Van de Vanter, and Michael Haupt. "Debugging at Full Speed". In: *Workshop on Dynamic Languages and Applications (Dyla)*. ACM, 2014, 2:1–2:13. DOI: 10.1145/2617548.2617550.

[113] Roberto Sebastiani and Silvia Tomasi. "Optimization in SMT with $LA(Q)$ Cost Functions". In: *Automated Reasoning*. Springer, 2012, pp. 484–498. ISBN: 978-3-642-31364-6. DOI: 10.1007/978-3-642-31365-3_38.

[114] Bastian Steinert, Lauritz Thamsen, Tim Felgentreff, and Robert Hirschfeld. "Object Versioning to Support Recovery Needs: Using Proxies to Preserve Previous Development States in Lively". In: *Dynamic Languages Symposium (DLS)*. ACM, 2014, pp. 113–124. DOI: 10.1145/2661088.2661093.

[115] Ivan E. Sutherland. "Sketchpad, A Man-Machine Graphical Communication System". PhD thesis. University of Cambridge, 1963. ISBN: 0-8240-4411-8.

[116] Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld. "Applying Data-driven Tool Development to Context-oriented Languages". In: *International Workshop on Context-Oriented Programming (COP)*. ACM, 2014, 1:1–1:7. DOI: 10.1145/2637066.2637067.

[117] Emina Torlak and Rastislav Bodík. "Growing solver-aided languages with rosette". In: *Symposium on New Ideas in Programming and Reflections on Software (Onward!)* ACM, 2013, pp. 135–152. DOI: 10.1145/2509578.2509586.

[118] Emina Torlak and Daniel Jackson. "Kodkod: A Relational Model Finder". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2007, pp. 632–647. DOI: 10.1007/978-3-540-71209-1_49.

[119] Brad Vander Zanden. "An Incremental Algorithm for Satisfying Hierarchies of Multi-way Dataflow Constraints". In: *ACM Transactions on Programming Languages and Systems* 18.1 (1996), pp. 30–72. DOI: 10.1145/225540.225543.

[120] David L Waltz. *Generating Semantic Description from Drawings of Scenes with Shadows*. Tech. rep. 271. MIT Artificial Intelligence Laboratory, 1972.

[121] Andrew Wiles. "Modular Elliptic Curves and Fermat's Last Theorem". In: *Annals of Mathematics* 141.3 (1995), pp. 443–551. DOI: 10.2307/2118559.

[122] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. "Z3-str: A Z3-based String Solver for Web Application Analysis". In: *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 114–124. DOI: 10.1145/2491411.2491456.

Part VI.

Appendix

Appendix A.

Full Formal Development

This chapter presents the complete semantics of our Babelsberg design that can be used to guide practical implementations [38]. It is self-contained and repeats relevant passages from Chapter 4 so there is no need to skip back to the main thesis.

This semantics is meant to be 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 in Chapter 3. First, Section A.1 presents the semantics of a simple imperative language that has only primitive reals, integers, and booleans. Second, Section A.2 adds the rules for method lookup and dispatch, identity constraints, and the interaction with the heap, and presents theorems for key properties, proofs for which have been previously published [39]. Third, Section A.3 presents the rules for constraints over collections.

Finally, Section 4.4 presents an executable form of the complete semantics that passes a language test suite. 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.

A.1. Primitive Types

We start with the 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, integers, and strings.

A.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`	`::=`	$\wp$ `e` | `C` $\wedge$ `C`
Expression	`e`	`::=`	`c` | `x` | `e` $\oplus$ `e`
Constant	`c`	`::=`	`true` | `false` | base type constants
Variable	`x`	`::=`	variable names
Value	`v`	`::=`	`c`

Table A.1.: Judgments and intuitions of semantic rules

Expression Evaluation	
$\mathtt{E} \vdash \mathtt{e} \Downarrow \mathtt{v}$	Expression e evaluates to value v in the context of environment E.
Constraint Solving	
$\mathtt{E} \models\!= \mathtt{C}$	Environment E represents a solution to constraint C
$\mathrm{stay}(\mathtt{x}{=}\mathtt{v}, \varrho) = \mathtt{C}$	Constraint C is a weak stay constraint for x to equal v
$\mathrm{stay}(\mathtt{E}, \varrho) = \mathtt{C}$	Constraint C is a conjunction of stay constraints on variables in environment E
Statement Evaluation	
$\mathtt{<E|C|s>} \longrightarrow \mathtt{<E'|C'>}$	Execution starting from configuration <E|C|s> ends in state <E′|C′>.

The language includes a set of boolean and base type constants (e.g., reals), ranged over by metavariable c. A finite set of operators on expressions is ranged over by $\oplus$. This includes operations on the reals such as + and $*$, a set of *predicate* operators ($=$ and $\neq$, $\leq$, $<$, $=$, and so on. It also includes a set of logical operators for combining boolean expressions (e.g., $\wedge$, $\vee$). The predicate operations are assumed to include at least an equality operator = for each primitive type in the language, and the logical operations are assumed to include at least conjunction $\wedge$. The syntax of this language does have some limitations as compared with that of a practical language — for example, there are only binary operators (not unary or ternary), and the result must have the same type as the arguments. We make these simplifications since the purpose of Babelsberg/PrimitiveTypes is to elucidate the semantics of such languages as a step toward Babelsberg/Objects, rather than to specify a real language.

For constraints, the symbol ϱ ranges over a finite and totally ordered set of constraint *priorities* and is assumed to include a bottom element `weak` and a top element `required`. While syntax requires the priority to be explicit, for simplicity we sometimes omit it in this semantics. A constraint with no explicit priority implicitly has the priority `required`. Finally, for simplicity we do not model read-only annotations in the formal semantics.

The syntax is thus that of a simple, standard imperative language except for the `always` and `once` statements, which declare constraints. An `always` constraint must hold for the rest of the programs execution, whereas a `once` constraint is satisfied by the solver and then retracted. Note that for simplicity this semantics implicitly gets stuck whenever the solver cannot satisfy a constraint, either due to an unsatisfiable constraint or due to the solver being unable to determine whether the constraint is satisfiable. In a practical implementation, we would likely want to differentiate between these cases, since it's useful if we can inform the programmer that the constraints are truly not satisfiable. We could also add standard exception handling to remove the unsatisfiable or unknown constraint and continue, but omit this here for simplicity.

A.1.2. Semantics

The semantics is defined by several judgments, given in Table A.1. These judgments depend on the notion of an *environment*, which is a partial function from program variables to program values. Metavariable E ranges over environments. When convenient we also view an environment as a set

of (program variable, program value) pairs. For each operator $\oplus$ in the language we assume the existence of a corresponding semantic function denoted $[\![\oplus]\!]$.

$$\mathtt{E} \vdash \mathtt{c} \Downarrow \mathtt{c} \tag{E-Const}$$

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

$$\frac{\mathtt{E} \vdash \mathtt{e_1} \Downarrow \mathtt{v_1} \qquad \mathtt{E} \vdash \mathtt{e_2} \Downarrow \mathtt{v_2} \qquad \mathtt{v_1}\ [\![\oplus]\!]\ \mathtt{v_2} = \mathtt{v}}{\mathtt{E} \vdash \mathtt{e_1}\ \oplus\ \mathtt{e_2} \Downarrow \mathtt{v}} \tag{E-Op}$$

Solving is represented by a call to the constraint solver, which we treat as a black box. The proposition $\mathtt{E} \models\!= \mathtt{C}$ 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).

Our helper judgments $\mathrm{stay}(\mathtt{x=v}, \varrho) = \mathtt{C}$ and $\mathrm{stay}(\mathtt{E}, \varrho) = \mathtt{C}$ define how to translate an environment into a source-level "stay" constraint. They take a priority as an argument, which is not used at this stage of the semantics, but will be used in all the later stages.

$$\frac{\begin{array}{c}\mathtt{E} = \{(\mathtt{x_1},\ \mathtt{v_1}),\ \ldots,\ (\mathtt{x}_n,\ \mathtt{v}_n)\} \\ \mathrm{stay}(\mathtt{x_1=v_1}, \varrho) = \mathtt{C_1} \cdots \mathrm{stay}(\mathtt{x}_n\mathtt{=v}_n, \varrho) = \mathtt{C}_n\end{array}}{\mathrm{stay}(\mathtt{E}, \varrho) = \mathtt{C_1}\ \wedge\ \cdots\ \wedge\ \mathtt{C}_n} \tag{StayEnv}$$

$$\mathrm{stay}(\mathtt{x=c}, \varrho) = \mathtt{weak\ \ x=c} \tag{StayConst}$$

The rules for evaluating statements are given below. 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.

$$\frac{\mathtt{E} \vdash \mathtt{e} \Downarrow \mathtt{v} \qquad \mathrm{stay}(\mathtt{E}, \varrho) = \mathtt{C}_s \qquad \mathtt{E'} \models\!= (\mathtt{C}\ \wedge\ \mathtt{C}_s\ \wedge\ \mathtt{x}\ =\ \mathtt{v})}{\mathtt{<E|C|x\ :=\ e>} \longrightarrow \mathtt{<E'|C>}} \tag{S-Asgn}$$

$$\frac{\mathrm{stay}(\mathtt{E}, \varrho) = \mathtt{C}_s \qquad \mathtt{E'} \models\!= (\mathtt{C}\ \wedge\ \mathtt{C}_s\ \wedge\ \mathtt{C}_0)}{\mathtt{<E|C|once\ C_0>} \longrightarrow \mathtt{<E'|C>}} \tag{S-Once}$$

$$\frac{\mathtt{<E|C|once\ C_0>} \longrightarrow \mathtt{<E'|C>} \qquad \mathtt{C'}\ =\ \mathtt{C}\ \wedge\ \mathtt{C}_0}{\mathtt{<E|C|always\ C_0>} \longrightarrow \mathtt{<E'|C'>}} \tag{S-Always}$$

$$\mathtt{<E|C|skip>} \longrightarrow \mathtt{<E|C>} \tag{S-Skip}$$

$$\frac{\mathtt{<E|C|s_1>} \longrightarrow \mathtt{<E'|C'>} \qquad \mathtt{<E'|C'|s_2>} \longrightarrow \mathtt{<E''|C''>}}{\mathtt{<E|C|s_1;s_2>} \longrightarrow \mathtt{<E''|C''>}} \tag{S-Seq}$$

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

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

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

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

A.2. Objects and Messages

The solver must now also handle objects which are modeled as simple records. To tame the power of the solver so that it adheres to the principles presented in 3, we add the two-phase solving, structural type checks, and inlining rules on constraints described in 3.2.4. These structural compatibility checks are assertions that are checked dynamically before sending the constraints involving objects to the solver, for example, checking whether a variable is bound to an object, and whether the object 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 an object with the expected fields is first assigned to a variable used in object constraints, just as a programmer would need to ensure that an object with the expected fields was assigned to a object-valued variable in a standard language.

The semantic rules are still mostly standard imperative rules, with the addition of rules to assert and maintain constraints. The semantics for this language, although omitting collections and cooperating constraint solvers, illustrates the key principles we want for an OCP language. The semantics still 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 [11]. Many practical examples use read-only variables, but this is not relevant for this semantics.

A.2.1. Syntax

The syntax is augmented to support creating instances of objects (both for value classes and ordinary classes) as well as method invocation. We introduce syntax for the method body.

Statement	s	::=	`skip` | `L := e` | `always C` | `once C` | `s;s`
			| `if e then s else s` | `while e do s`
Constraint	C	::=	$\wp$ `e` | `C` $\wedge$ `C`
Expression	e	::=	`v` | `L` | `e` $\oplus$ `e` | `I`
			| $\texttt{e.l(e}_1\texttt{,...,e}_n\texttt{)}$ | `o` | `new o` | `D`
Identity	I	::=	`e == e`
Object Literal	o	::=	$\{\texttt{l}_1\texttt{:e}_1\texttt{,...,l}_n\texttt{:e}_n\}$
L-Value	L	::=	`x` | `e.l`
Constant	c	::=	`true` | `false` | `nil` | base type constants
Variable	x	::=	variable names
Label	l	::=	record label names
Reference	r	::=	references to heap records
Dereference	D	::=	`H(e)`
Method Body	b	::=	`s; return e` | `return e`
Value	v	::=	`c` | `r` | $\{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}$

The syntax elements `c`, $\wp$, and $\oplus$ are mostly unchanged. We add the operator `==`, which tests for identity — for primitive values this behaves the same as `=`. We add labels for naming fields, and syntax for referencing them. Since we now have a heap, in the syntax, we treat `H` as a keyword used for dereferencing. Source programs will not use expressions of the form `H(e)`, but they are introduced as part of constraints given to the solver, which we assume will treat `H` as an uninterpreted function. We also assume that the solver supports records and record equality, which we also denote with the = operator. We furthermore assume that the set of variable names ranged over by metavariable `x` includes the name `self`.

A.2.2. Semantics

The semantics now include a global $\mathbb{E}$ and a heap $\mathbb{H}$, in addition to method local scopes $\mathbb{S}$. The first is a function that maps global variable names to values, the second is a function that maps mutable references to "objects" of the form $\{\texttt{l}_1\texttt{:v}_1\texttt{,...,l}_n\texttt{:v}_n\}$, and the third now maps local variable names to global variable names. When convenient, we also treat both $\mathbb{E}$ and $\mathbb{H}$ as a set 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 single conjunction referred to as $\mathbb{I}$.

This semantics requires updated and additional judgments given in Table A.4. The rules for expression evaluation still work on the local scope, but since a local scope is now only a mapping from local names to globally unique names, the rules are augmented to then look up the value in the global environment. Method lookup creates new local environments, and constraints are translated to translate local variable names to global variable names for the solver. The solver still returns a global environment, and does not know about the local environments and their mapping into the global environment. In addition, the constraint stores `I` and `C` are replaced with $\mathbb{I}$ and $\mathbb{C}$, respectively. These now store constraints in pairs with the local environment they were created in. For readability, we now write $\mathbb{H}$ instead of `H`, so all the global stores are written in the same font.

The judgment for method lookup is opaque: our semantics does not depend on how method lookup is performed. After lookup we create a fresh scope with the evaluated arguments bound to the parameters of the method, the rule for which is given below:

Method Invocation	
$lookup(\mathtt{v},\mathtt{l}) = (\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{b})$	Lookup of method `l` in the object or value `v` returns the formal parameter names $\mathtt{x}_1$ through $\mathtt{x}_n$ and the method body `b`
$enter(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\ \mathbb{I},\mathtt{v},\mathtt{x}_1 \cdots \mathtt{x}_n,\mathtt{e}_1 \cdots \mathtt{e}_n) = (\mathbb{E}',\mathbb{S}_m,\mathbb{H}',\mathbb{C}',\mathbb{I}')$	Invoking a method on `v` with argument names $\mathtt{x}_1$ through $\mathtt{x}_n$ and arguments $\mathtt{e}_1$ through $\mathtt{e}_n$ constructs the method scope $\mathbb{S}_m$ and may update the global state.
Expression Evaluation	
$\mathtt{<}\mathbb{E}\mid\mathbb{S}\mid\mathbb{H}\mid\mathbb{C}\mid\mathbb{I}\mid\mathtt{e}\mathtt{>} \Downarrow \mathtt{<}\mathbb{E}'\mid\mathbb{H}'\mid\mathbb{C}'\mid\mathbb{I}'\mid\mathtt{v}\mathtt{>}$	Evaluating expression `e` produces the value `v` and updated state $\mathbb{E}'$, $\mathbb{H}'$, $\mathbb{C}'$, and $\mathbb{I}'$
Typechecking	
$\mathbb{E};\mathbb{H} \vdash \mathtt{e} : \mathtt{T}$	Expression `e` has type `T` in the context of environment $\mathbb{E}$ and heap $\mathbb{H}$
$\mathbb{E};\mathbb{H} \vdash \mathtt{C}$	Constraint `C` is well formed in the context of environment $\mathbb{E}$ and heap $\mathbb{H}$
Constraint Solving	
$\mathbb{E};\mathbb{H} \models\!= \mathtt{C}$	Environment $\mathbb{E}$ and heap $\mathbb{H}$ represent a solution to constraint `C`
$\text{stay}(\mathbb{E}, \varrho) = \mathtt{C}$	Constraint `C` is a conjunction of stay constraints on variables in environment $\mathbb{E}$
$\text{stay}(\mathbb{H}, \varrho) = \mathtt{C}$	Constraint `C` is a conjunction of stay constraints on objects in heap $\mathbb{H}$
$\text{stay}(\mathtt{x=c}, \varrho) = \mathtt{C}$	Constraint `C` is a weak stay constraint for `x` to equal `c`
$\text{stay}(\mathtt{x=r}, \varrho) = \mathtt{C}$	Constraint `C` is a stay constraint with priority ϱ for `x` to equal `r`
$\text{stay}(\mathtt{r=o}, \varrho) = \mathtt{C}$	Constraint `C` is a required stay constraint for reference `r` to refer to the object `o`
$\mathtt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\mathtt{e}\mathtt{>} \rightsquigarrow \mathtt{<}\mathbb{E}',\mathtt{e}_C,\mathtt{e}'\mathtt{>}$	Inlining expression `e` in $\mathbb{S}$ is equivalent to `e'` in $\mathbb{E}$ if $\mathtt{e}_C$ evaluates to true.
$\mathtt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{I},\mathbb{C}\ \mathtt{>} \rightsquigarrow \mathtt{<}\mathbb{E}',\mathtt{C}\mathtt{>}$	Re-inlining the constraint store $\mathbb{C}$ returns a constraint `C`
$\mathtt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I}\ \mathtt{>} \rightsquigarrow \mathtt{<}\mathbb{E}',\mathtt{C}\mathtt{>}$	Re-inlining the constraint store $\mathbb{I}$ returns a constraint `C`
Statement Evaluation	
$\mathtt{<}\mathbb{E}\mid\mathbb{S}\mid\mathbb{H}\mid\mathbb{C}\mid\mathbb{I}\mid\mathtt{s}\mathtt{>} \longrightarrow \mathtt{<}\mathbb{E}'\mid\mathbb{S}'\mid\mathbb{H}'\mid\mathbb{C}'\mid\mathbb{I}'\mathtt{>}$	Execution starting from state $\mathtt{<}\mathbb{E}\mid\mathbb{S}\mid\mathbb{H}\mid\mathbb{C}\mid\mathbb{I}\mid\mathtt{s}\mathtt{>}$ ends in $\mathtt{<}\mathbb{E}'\mid\mathbb{S}'\mid\mathbb{H}'\mid\mathbb{C}'\mid\mathbb{I}'\mathtt{>}$
Helper Rules	
$\text{solve}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\varrho,\mathtt{e}) = \mathbb{E}',\mathbb{H}'$	Solving identity constraints and ϱ `e` yields new environment and heap $\mathbb{E}'$ and $\mathbb{H}'$
$\text{solve}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{I},\mathbb{C},\varrho,\mathtt{e}) = \mathbb{E}',\mathbb{H}'$	Solving value constraints and ϱ `e` yields new environment and heap $\mathbb{E}'$ and $\mathbb{H}'$

Table A.4.: Judgments and intuitions of semantic rules for objects

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

Expression evaluation can now have effects on the global environment, heap, and the constraint stores (but not on the local scope), because calling methods is possible in expressions.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We add a typing rule for dereferences. We assume dereferences will not appear in source programs, but only in expressions that have been generated by our inlining judgment. This ensures that the inlined expressions still typecheck, and simplifies the rules for constraint solving.

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{e_1 : PrimitiveType} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{e_2 : PrimitiveType}}{\mathbb{E};\mathbb{H} \vdash \mathtt{e_1 \ \oplus \ e_2 : PrimitiveType}} \quad \text{(T-Op)}$$

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

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

$$\frac{\mathbb{E};\mathbb{H} \vdash \mathtt{C_1 : T} \qquad \mathbb{E};\mathbb{H} \vdash \mathtt{C_2 : T}}{\mathbb{E};\mathbb{H} \vdash \mathtt{C_1 \ \wedge \ C_2}} \quad \text{(T-Conjunction)}$$

The rules for creating stay constraints are simply extended to include rules for stay constraints on instances of objects, and adapted to this formalism's environment and heap.

$$\text{stay}(\mathtt{x{=}c}, \varrho) = \mathtt{weak \ x{=}c} \quad \text{(StayConst)}$$

$$\text{stay}(\mathtt{x{=}r}, \varrho) = \varrho \ \mathtt{x{=}r} \quad \text{(StayRef)}$$

$$\frac{\mathtt{x_1}\text{ fresh} \cdots \ \mathtt{x_n}\text{ fresh} \qquad \text{stay}(\mathtt{x_1{=}v_1}, \varrho) = \mathtt{C_1} \cdots \text{stay}(\mathtt{x_n{=}v_n}, \varrho) = \mathtt{C_n} \quad \mathtt{C{=}(}\varrho \ \mathtt{x \ = \ \{l_1{:}x_1,\ldots,l_n{:}x_n\}) \ \wedge \ C_1 \ \wedge \cdots \ \wedge \ C_n}}{\text{stay}(\mathtt{x \ = \ \{l_1{:}v_1,\ldots,l_n{:}v_n\}}, \varrho) = \mathtt{C}} \quad \text{(StayRecord)}$$

In StayRecord we do not allow records to change their when solving for value constraints, record structure can only change through assignments.

$$\frac{\mathtt{x_1}\text{ fresh} \cdots \ \mathtt{x_n}\text{ fresh} \qquad \text{stay}(\mathtt{x_1{=}v_1}, \varrho) = \mathtt{C_1} \cdots \text{stay}(\mathtt{x_n{=}v_n}, \varrho) = \mathtt{C_n} \quad \mathtt{C{=}(required \ H(r){=}\{l_1{:}x_1,\ldots,l_n{:}x_n\}) \ \wedge \ C_1 \ \wedge \cdots \ \wedge \ C_n}}{\text{stay}(\mathtt{r \ = \ \{l_1{:}v_1,\ldots,l_n{:}v_n\}}, \varrho) = \mathtt{C}} \quad \text{(StayObject)}$$

$$\frac{\mathbb{E} = \mathtt{\{(x_1, \ v_1),\ldots,(x_n, \ v_n)\}} \qquad \text{stay}(\mathtt{x_1{=}v_1}, \varrho) = \mathtt{C_1} \cdots \text{stay}(\mathtt{x_n{=}v_n}, \varrho) = \mathtt{C_n}}{\text{stay}(\mathbb{E}, \varrho) = \mathtt{C_1 \ \wedge \cdots \ \wedge \ C_n}} \quad \text{(StayEnv)}$$

$$\frac{\mathbb{H} = \mathtt{\{(r_1, \ o_1),\ldots,(r_n, \ o_n)\}} \qquad \text{stay}(\mathtt{r_1{=}o_1}, \varrho) = \mathtt{C_1} \cdots \text{stay}(\mathtt{r_n{=}o_n}, \varrho) = \mathtt{C_n}}{\text{stay}(\mathbb{H}, \varrho) = \mathtt{C_1 \ \wedge \cdots \ \wedge \ C_n}} \quad \text{(StayHeap)}$$

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

In our inlining rules, 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. Since expressions are actually evaluated during inlining, if they do modify the heap, evaluation stops and the rules fail.

$$\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{c>} \rightsquigarrow \texttt{<}\mathbb{E},\texttt{true},\texttt{c>} \tag{I-Const}$$

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

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

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

The I-Field rule translates each expression of the form `e.l`, where `e` evaluates to a heap reference `r`, into `H(e').l` (recursively translating `e` into `e'` through further inlining). These judgments further ensure that the "prefix" `e` of an l-value `e.l` is unchanged through the addition of the constraint `e'=r`; this is necessary to ensure that updates to the value of `e.l` are deterministic. Note that, because we are recursively inlining the expression before the label, we also recursively add required stays on any prior field accesses. Thus, for example, `a.b.c = 1` would be translated by recursively inlining and returning `a = r`$_a$ `∧ H(a).b = r`$_b$ `∧ H(H(a).b).c = 1`.

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{>} \\ \texttt{<}\mathbb{E}'|\mathbb{S}|\mathbb{H}|\mathbb{C}|\mathbb{I}|\texttt{e>} \Downarrow \texttt{<}\mathbb{E}''|\mathbb{H}|\mathbb{C}|\mathbb{I}|\{\texttt{l}_1\texttt{:v}_1,\ldots,\texttt{l}_n\texttt{:v}_n\}\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e.l>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{.l>}} \tag{I-ValueField}$$

$$\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{r>} \rightsquigarrow \texttt{<}\mathbb{E},\texttt{true},\texttt{r>} \tag{I-Ref}$$

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

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

The I-Identity rule ensures that identities cannot change in value constraint expressions. Because we inline the expressions on either side, these, through I-Field and I-ValueField, are forced to stay as they are up to the last part. Note that that inlined identity constraints do not type (in S-Once), even if both operands have the same type — this prevents adding identity constraints mixed with value constraints.

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

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

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

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

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

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

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

The rules below are helpers for use in solving constraints. They generate the inlined constraint expression from e, re-inline the constraints from the identity respectively constraint stores, and to generate the stay constraints and solve for these. In the first phase, we use the first judgment above

to inline the identity constraint store and generate the stays on the heap and environment such that the solve can 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 use the second judgment above generate constraints for the non-identity constraint store, and we typecheck under the new environment before solving.

$$\frac{\begin{array}{c}\text{stay}(\mathbb{E}, \texttt{weak}) = \texttt{C}_{E_s} \qquad \text{stay}(\mathbb{H}, \texttt{weak}) = \texttt{C}_{H_s} \\ \texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I}\texttt{ > } \rightsquigarrow \texttt{<}\mathbb{E}_i,\texttt{C}_i\texttt{>} \\ \texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}\texttt{> } \rightsquigarrow \texttt{<}\mathbb{E}_e,\texttt{e}_c,\texttt{e}'\texttt{> } \qquad \mathbb{E}';\mathbb{H}' \models\!\!= \texttt{C}_i \ \wedge\ \varrho\ \texttt{e}_C\end{array}}{\text{solve}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\varrho,\texttt{e}) = \mathbb{E}',\mathbb{H}'}$$

(SolveIdentityConstraints)

$$\frac{\begin{array}{c}\text{stay}(\mathbb{E}, \texttt{required}) = \texttt{C}_{E_s} \qquad \text{stay}(\mathbb{H}, \texttt{required}) = \texttt{C}_{H_s} \\ \texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{I},\mathbb{C}\texttt{ > } \rightsquigarrow \texttt{<}\mathbb{E}_c,\texttt{C}_c\texttt{>} \\ \texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e}\texttt{> } \rightsquigarrow \texttt{<}\mathbb{E}_e,\texttt{e}_c,\texttt{e}'\texttt{> } \qquad \mathbb{E}_e;\mathbb{H} \vdash \texttt{e}' : \texttt{T} \qquad \mathbb{E}';\mathbb{H}' \models\!\!= \texttt{C}_c \ \wedge\ \varrho\ \texttt{e}_C\end{array}}{\text{solve}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{I},\mathbb{C},\varrho,\texttt{e}) = \mathbb{E}',\mathbb{H}'}$$

(SolveValueConstraints)

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

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

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

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

Note that the above rule can only be used in constraint-construction mode if $\mathbb{H}$=$\mathbb{H}'$. We also do not allow the use of the following rules for `once` and `always` in constraint-construction mode, because the inlining rules disallow updating the constraint store and heap. As discussed previously (Section 3.2.6), a practical implementation we might want to support benign side effects in methods that are invoked by constraint expressions, including methods that themselves create new constraints; but this is not modeled by this semantics.

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

(S-OnceIdentity)

Note that the above rule uses the inlining judgment to ensure that the expressions adhere to our restrictions on what kind of expressions can appear in constraints. Expressions that update the heap, for example, will be rejected in the inlining rules.

$$\frac{\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|once } e_0 \texttt{ == } e_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{ >} \quad \mathbb{I}' = \mathbb{I} \bigcup \{(\mathbb{S}, e_0 == e_1)\}}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|always } e_0 \texttt{ == } e_1\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}'\texttt{>}} \text{(S-AlwaysIdentity)}$$

$$\frac{c_0 = \varsigma\, e \qquad \text{solve}(\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{I},\mathbb{C},\varsigma,e) = \mathbb{E}',\mathbb{H}'}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|once } c_0\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{ >}} \text{(S-Once)}$$

$$\frac{\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|once } c_0\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{ >} \quad \mathbb{C}' = \mathbb{C} \bigcup \{(\mathbb{S}, c_0)\}}{\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|always } c_0\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}\texttt{ >}} \text{(S-Always)}$$

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

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

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

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

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

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

A.2.3. Key Properties

In prior publications, we have presented and proven two key properties of our formalism for a simpler language, called Babelsberg/UID [38, 40]. Here we state the same two key theorems about our formalism, updated for the Babelsberg/Objects language presented here. The first theorem formalizes the idea that any solution to a value constraint preserves the structures of the objects on the environment and heap:

Theorem 1. *(Structure Preservation) If* $\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|}(\texttt{once}\texttt{|}\texttt{always})\ c_0\texttt{>} \longrightarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{S}'\texttt{|}\mathbb{H}'\texttt{|}\mathbb{C}'\texttt{|}\mathbb{I}'\texttt{>}$ *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* $\langle \mathbb{E} | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{L := e} \rangle \longrightarrow \langle \mathbb{E}_1 | \mathbb{S}_2 | \mathbb{H}_1 | \mathbb{C}_1 | \mathbb{I}_1 \rangle$ *and* $\langle \mathbb{E} | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{L := e} \rangle \longrightarrow \langle \mathbb{E}_2 | \mathbb{S}_2 | \mathbb{H}_2 | \mathbb{C}_2 | \mathbb{I}_2 \rangle$ *and* $\langle \mathbb{E} | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{I} \rangle \Downarrow \langle \texttt{true} | a | n | d | \rangle x$ *in dom(*$\mathbb{E}$*), then* $\mathbb{E}_1 ; \mathbb{H}_1 \vdash x : T$ *and* $\mathbb{E}_2 ; \mathbb{H}_2 \vdash x : T$.

A.3. Collections of Objects

In this final extension to Babelsberg/Objects, we support a limited number of predicates on collections. These additional rules illustrate the facilities to support higher-order functions in this design. The semantics are a straight extension from the full object-oriented language, and we present only the additional rules and syntax.

A.3.1. Syntax

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

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

A.3.2. Semantics

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

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

$$\frac{\langle \mathbb{E} | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{e}_l \rangle \Downarrow \langle \mathbb{E}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' | \texttt{c} \rangle \qquad \mathit{asLabel}(\texttt{c}) = \texttt{l} \quad \langle \mathbb{E}' | \mathbb{S}' | \mathbb{H}' | \mathbb{C}' | \mathbb{I}' | \texttt{e.l} \rangle \Downarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' | \texttt{v} \rangle}{\langle \mathbb{E} | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{e[e}_l\texttt{]} \rangle \Downarrow \langle \mathbb{E}'' | \mathbb{H}'' | \mathbb{C}'' | \mathbb{I}'' | \texttt{v} \rangle} \quad \text{(E-ExpField)}$$

$$\frac{\langle \mathbb{E}, \mathbb{S}, \mathbb{H}, \mathbb{C}, \mathbb{I}, \texttt{e}_l \rangle \rightsquigarrow \langle \mathbb{E}', \texttt{e}_{C_l}, \texttt{e}'_l \rangle \quad \langle \mathbb{E}' | \mathbb{S} | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{e}_l \rangle \Downarrow \langle \mathbb{E}'' | \mathbb{H} | \mathbb{C} | \mathbb{I} | \texttt{c} \rangle \qquad \mathit{asLabel}(\texttt{c}) = \texttt{l} \quad \langle \mathbb{E}'', \mathbb{S}, \mathbb{H}, \mathbb{C}, \mathbb{I}, \texttt{e.l} \rangle \rightsquigarrow \langle \mathbb{E}''', \texttt{e}_C, \texttt{e}' \rangle}{\langle \mathbb{E}, \mathbb{S}, \mathbb{H}, \mathbb{C}, \mathbb{I}, \texttt{e[e}_l\texttt{]} \rangle \rightsquigarrow \langle \mathbb{E}''', \texttt{e}_{C_l} \wedge \texttt{e}_C \wedge \texttt{e}'_l\texttt{=c}, \texttt{e}' \rangle} \quad \text{(I-ExpField)}$$

Table A.5.: Judgments and intuitions of additional and changed semantic rules

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

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

We define a helper judgments to inline collection predicates:

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

This helper rule sets up the required equalities for all the arguments and the receiver, and is essentially the same as I-Call. As an addition, it limits any inlining to collection predicates that

return a constant as a final statement. In the two rules that follow, this constant is further limited to be either `true` or `false`.

$$\frac{\mathit{preparePredicate}(\mathbb{E}, \mathbb{S}, \mathbb{H}, \mathbb{C}, \mathbb{I}, \texttt{e.Ⓟ (e}_1\texttt{,...,e}_n\texttt{)}) = (\mathbb{E}', \mathbb{S}', \texttt{s; return c}, \texttt{e}_0) \quad \texttt{c = true} \quad \texttt{<}\mathbb{E}'\texttt{,}\mathbb{S}'\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,s>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_1\texttt{,e}_C\texttt{,e}_D\texttt{>}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,e.Ⓟ (e}_1\texttt{,...,e}_n\texttt{)>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_0\texttt{∧e}_1\texttt{,e}_C\texttt{>}} \text{(I-PositivePredicate)}$$

$$\frac{\mathit{preparePredicate}(\mathbb{E}, \mathbb{S}, \mathbb{H}, \mathbb{C}, \mathbb{I}, \texttt{e.Ⓟ (e}_1\texttt{,...,e}_n\texttt{)}) = (\mathbb{E}', \mathbb{S}', \texttt{s; return c}, \texttt{e}_0) \quad \texttt{c = false} \quad \texttt{<}\mathbb{E}'\texttt{,}\mathbb{S}'\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,s>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_1\texttt{,e}_C\texttt{,e}_D\texttt{>}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,e.Ⓟ (e}_1\texttt{,...,e}_n\texttt{)>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_0\texttt{∧e}_1\texttt{,e}_C \texttt{∧ e}_D\texttt{>}} \text{(I-NegativePredicate)}$$

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

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

$$\frac{\mathbb{S}(\texttt{x})\uparrow \quad \mathbb{E}(\texttt{x}_g)\uparrow \quad \texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v>} \quad \texttt{<}\mathbb{E}'\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,e>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_0\texttt{,e}'\texttt{>} \quad \mathbb{S}' = \mathbb{S} \bigcup \{(\texttt{x}, \texttt{x}_g)\} \quad \mathbb{E}''' = \mathbb{E}'' \bigcup \{(\texttt{x}_g, \texttt{v})\}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,x := e>} \rightsquigarrow \texttt{<}\mathbb{E}'''\texttt{,}\mathbb{S}'\texttt{,e}_0\texttt{∧e}'\texttt{=v∧x}_g\texttt{=v,true,false>}} \text{(I-AsgnNewLocal)}$$

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

$$\frac{\mathbb{S}(\texttt{x}) = \texttt{x}_g \quad \mathbb{E}(\texttt{x}'_g)\uparrow \quad \texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|v>} \quad \texttt{<}\mathbb{E}'\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,e>} \rightsquigarrow \texttt{<}\mathbb{E}''\texttt{,e}_0\texttt{,e}'\texttt{>} \quad \mathbb{S}' = \mathbb{S} \setminus \{\texttt{x}, \texttt{x}_g\} \quad \mathbb{S}'' = \mathbb{S}' \bigcup \{(\texttt{x}, \texttt{x}'_g)\} \quad \mathbb{E}''' = \mathbb{E}'' \bigcup \{(\texttt{x}'_g, \texttt{v})\}}{\texttt{<}\mathbb{E}\texttt{,}\mathbb{S}\texttt{,}\mathbb{H}\texttt{,}\mathbb{C}\texttt{,}\mathbb{I}\texttt{,x := e>} \rightsquigarrow \texttt{<}\mathbb{E}'''\texttt{,}\mathbb{S}''\texttt{,e}_0\texttt{∧e}'\texttt{=v∧x}'_g\texttt{=v,true,false>}} \text{(I-AsgnLocal)}$$

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

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

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_1\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}',\mathbb{S}',\texttt{e}_1,\texttt{e}_{C1},\texttt{e}_{D1}\texttt{>} \\ \texttt{<}\mathbb{E}',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\mathbb{S}'',\texttt{e}_2,\texttt{e}_{C2},\texttt{e}_{D2}\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s}_1\texttt{;}\texttt{s}_2\texttt{>} \rightsquigarrow \texttt{<}\mathbb{E}'',\mathbb{S}'',\texttt{e}_1 \wedge \texttt{e}_2,\texttt{e}_{C1} \wedge \texttt{e}_{C2},\texttt{e}_{D1} \vee \texttt{e}_{D2}\texttt{>}} \quad \text{(I-Seq)}$$

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

$$\frac{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{>}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{if e then return true else s>} \rightsquigarrow \texttt{<}\mathbb{E},\mathbb{S},\texttt{e}_C,\texttt{true},\texttt{e}'\texttt{>}} \quad \text{(I-IfThenReturnTrue)}$$

$$\frac{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e>} \rightsquigarrow \texttt{<}\mathbb{E}',\texttt{e}_C,\texttt{e}'\texttt{>}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{if e then return false else s>} \rightsquigarrow \texttt{<}\mathbb{E},\mathbb{S},\texttt{e}_C,\texttt{e}'\texttt{=false},\texttt{false>}} \quad \text{(I-IfThenReturnFalse)}$$

We only support if-clauses used as early returns in this extended inlining mode. As described in Section 3.3.1, if the early return would return `true`, the inlined conditional is used in a disjunction, otherwise it is used in a conjunction.

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|true>} \\ \texttt{<}\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_0,\texttt{e}'\texttt{>} \\ \texttt{<}\mathbb{E}'',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{s>} \rightsquigarrow \texttt{<}\mathbb{E}''',\mathbb{S}',\texttt{e}_1,\texttt{e}_{C_1},\texttt{e}_{D_1}\texttt{>} \\ \texttt{<}\mathbb{E}''',\mathbb{S}',\mathbb{H},\mathbb{C},\mathbb{I},\texttt{while e do s>} \rightsquigarrow \texttt{<}\mathbb{E}'''',\mathbb{S}'',\texttt{e}_r,\texttt{e}_{C_r},\texttt{e}_{D_r}\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{while e do s>} \rightsquigarrow \texttt{<}\mathbb{E}'''',\mathbb{S}'',\texttt{e}_0 \wedge \texttt{e}' \wedge \texttt{e}_1 \wedge \texttt{e}_r,\texttt{e}_{C_0} \wedge \texttt{e}_{C_r},\texttt{e}_{D_0} \vee \texttt{e}_{D_r}\texttt{>}} \quad \text{(I-WhileDo)}$$

$$\frac{\begin{array}{c}\texttt{<}\mathbb{E}\texttt{|}\mathbb{S}\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|e>} \Downarrow \texttt{<}\mathbb{E}'\texttt{|}\mathbb{H}\texttt{|}\mathbb{C}\texttt{|}\mathbb{I}\texttt{|false>} \\ \texttt{<}\mathbb{E}',\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{e>} \rightsquigarrow \texttt{<}\mathbb{E}'',\texttt{e}_0,\texttt{e}'\texttt{>}\end{array}}{\texttt{<}\mathbb{E},\mathbb{S},\mathbb{H},\mathbb{C},\mathbb{I},\texttt{while e do s>} \rightsquigarrow \texttt{<}\mathbb{E}'',\mathbb{S},\texttt{e}_0 \wedge \texttt{e}'\texttt{=false},\texttt{true},\texttt{false>}} \quad \text{(I-WhileSkip)}$$

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

⁂

There is a noteworthy problem with these rules: they may generate constraints that are too strong. Consider the following method, which tests if either at least one element in the array is larger than ten, or else all elements are negative:

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

```
  return true
end

always array.some_or_none()
```

Here, the constraint would be satisfied if:

$$\exists x \in \texttt{array}.x > 10 \vee \forall x \in \texttt{array}.x < 0$$

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

$$\forall x \in \texttt{array}.x < 0$$

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

Appendix B.

Benchmarks

This section gives the source code listings for the benchmarks that we used to evaluate the performance of our OCP implementations. (These are also included in the software archive that comes with this thesis, the filename for each listing is given in the title.)

Read Access to Constrained Variables

The following code listings show how we measured read access performance in Ruby, JavaScript, and Squeak/Smalltalk.

```
require "mybenchmark"
require "libcassowary"

class MockObject
  attr_accessor :a, :b, :c, :d, :e

  def initialize
    @a,@b,@c,@d,@e = 1,1,1,1,1
  end
end

obj, constraint = nil, nil

Resetter = Proc.new do |label|
  obj = MockObject.new
  if label.start_with? "Constrained"
    constraint.disable if constraint
    constraint = always do
      obj.a == 1 &&
        obj.b == 1 &&
        obj.c == 1 &&
        obj.d == 1 &&
        obj.e == 1
    end
    constraint.disable if label.end_with? "(disabled)"
  end
end

class SetupSuite
  def warming(label, *args)
    Resetter[label]
  end
  def warmup_stats(*); end
  alias_method :add_report, :warmup_stats
  alias_method :running, :warming
end
suite = SetupSuite.new

benchmark = Proc.new { |t| t.times { obj.a + obj.b + obj.c + obj.d + obj.e } }

Benchmark.ips do |x|
  x.config(:suite => suite)
  x.report('Unconstrained Read', &benchmark)
  x.report('Constrained Read', &benchmark)
  x.report('Constrained Read (disabled)', &benchmark)
  x.compare!
end
```

Language: Ruby, Filename: benchmark-read-access/benchmark.rb

```
var suite = new Benchmark.Suite;

benchf = function() {
   obj.a + obj.b + obj.c + obj.d + obj.e
}
benchmarks = {
   "Unconstrained Read": [initf = function() {
      obj = {a: 1, b: 1, c:1, d:1, e:1};
   }, benchf],
   "Properties Read": [function() {
      obj = {
         get a() { return this.$$a }, $$a: 0,
         get b() { return this.$$b }, $$b: 0,
         get c() { return this.$$c }, $$c: 0,
         get d() { return this.$$d }, $$d: 0,
```

```
        get e() { return this.$$e }, $$e: 0,
      }
   }, benchf],
   "Constrained Read": [constrainf = function() {
      initf()
      constraint = bbb.always({solver: new ClSimplexSolver(), ctx: {obj: obj}}, function () {
         return obj.a == 1 &&
            obj.b == 1 &&
            obj.c == 1 &&
            obj.d == 1 &&
            obj.e == 1
      });
   }, benchf],
   "Constrained Read (disabled)": [disablef = function() {
      constrainf()
      constraint.disable();
   }, benchf],
   "Constrained Read (disabled, unconstrained)": [function() {
      disablef();
      bbb.unconstrainAll(obj);
   }, benchf]
}

for (var k in benchmarks) {
   suite.add(k, benchmarks[k][1], {setup: benchmarks[k][0]})
}
```

Language: JavaScript, Filename: benchmark-read-access/benchmark.js

```
'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 11:09:18 am'!
Object subclass: #ConstraintMockObject
 instanceVariableNames: 'a b c d e'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Constraints-Benchmarks'!

"... accessors and initializers for the mock object ..."

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 10:53:31 am'!
Benchmark subclass: #ConstraintsBenchmarks
 instanceVariableNames: 'obj constrObj constraints'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Constraints-Benchmarks'!

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:42'!
benchConstrainedReadAccess

 1000 timesRepeat: [
  constrObj a + constrObj b + constrObj c + constrObj d + constrObj e
 ]! !

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:47'!
benchConstrainedReadAccessDisabled

 constraints do: #disable.
 1000 timesRepeat: [
  constrObj a + constrObj b + constrObj c + constrObj d + constrObj e
 ]! !

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:39'!
benchReadAccess

 1000 timesRepeat: [
  obj a + obj b + obj c + obj d + obj e
 ]! !


!ConstraintsBenchmarks methodsFor: 'running' stamp: 'tfel 9/23/2015 15:46'!
setUp

 obj := ConstraintMockObject new.
 constrObj := ConstraintMockObject new.
 constraints := OrderedCollection new.
 constraints add: [constrObj a = 1] alwaysTrue.
 constraints add: [constrObj b = 1] alwaysTrue.
 constraints add: [constrObj c = 1] alwaysTrue.
 constraints add: [constrObj d = 1] alwaysTrue.
 constraints add: [constrObj e = 1] alwaysTrue.
! !
```

Language: Squeak Changesets, Filename: benchmark-read-access/benchmark.st

Write Access to Constrained Variables

The following code listings show how we measured write access performance in Ruby, JavaScript, and Squeak/Smalltalk.

```
require "mybenchmark"
require "libcassowary"

class MockObject
 attr_accessor :a, :b, :c, :d, :e

 def initialize(a=1,b=1,c=1,d=1,e=1)
  @a,@b,@c,@d,@e = a,b,c,d,e
 end
```

```
end

obj, constraint = nil, nil

Resetter = Proc.new do |label|
  obj = MockObject.new
  if label.start_with? "Constrained"
    constraint.disable if constraint
    constraint = always do
      obj.a >= 1 &&
        obj.b >= 1 &&
        obj.c >= 1 &&
        obj.d >= 1 &&
        obj.e >= 1
    end
    constraint.disable if label.end_with? "(disabled)"
  end
end

class SetupSuite
  def warming(label, *args)
    Resetter[label]
  end
  def warmup_stats(*); end
  alias_method :add_report, :warmup_stats
  alias_method :running, :warming
end
suite = SetupSuite.new

benchmark = Proc.new { |t| t.times { obj.a += 1; obj.b += 1; obj.c += 1; obj.d += 1; obj.e += 1 } }

Benchmark.ips do |x|
  x.config(:suite => suite)
  x.report('Unconstrained Write', &benchmark)
  x.report('Constrained Write', &benchmark)
  x.report('Constrained Write (disabled)', &benchmark)
  x.report('Constrained Write (edit)') do |times|
    class MyStream
      def initialize(times)
        @times = times
      end
      def next
        raise StopIteration if @times == 0
        o = MockObject.new(@times, @times, @times, @times, @times)
        @times -= 1
        o
      end
    end
    edit(stream: MyStream.new(times), accessors: ["a", "b", "c", "d", "e"]) { obj }
  end
  x.compare!
end
```

Language: Ruby, Filename: benchmark-write-access/benchmark.rb

```
var suite = new Benchmark.Suite;

var suite = new Benchmark.Suite;

benchf = function() {
   obj.a += 1;
   obj.b += 1;
   obj.c += 1;
   obj.d += 1;
   obj.e += 1;
}
benchmarks = {
   "Unconstrained Write": [initf = function() {
      obj = {a: 1, b: 1, c:1, d:1, e:1};
   }, benchf],
   "Properties Write": [function() {
      obj = {
         get a() { return this.$$a }, $$a: 0,
         get b() { return this.$$b }, $$b: 0,
         get c() { return this.$$c }, $$c: 0,
         get d() { return this.$$d }, $$d: 0,
         get e() { return this.$$e }, $$e: 0,
      }
   }, benchf],
   "Constrained Write": [constrainf = function() {
      initf();
      constraint = bbb.always({solver: new ClSimplexSolver(), ctx: {obj: obj}}, function () {
         return obj.a >= 1 &&
            obj.b >= 1 &&
            obj.c >= 1 &&
            obj.d >= 1 &&
            obj.e >= 1
      });
   }, benchf],
   "Constrained Write (disabled)": [disablef = function() {
      constrainf();
      constraint.disable();
   }, benchf],
   "Constrained Write (disabled, unconstrained)": [function() {
      disablef();
      bbb.unconstrainAll(obj);
   }, benchf],
   "Constrained Write (edit)": [function() {
      constrainf();
      cb = bbb.edit(obj, ["a", "b", "c", "d", "e"]);
   }, function() {
      cb([obj.a + 1, obj.b + 1, obj.c + 1, obj.d + 1, obj.e + 1]);
   }]
}

for (var k in benchmarks) {
   suite.add(k, benchmarks[k][1], {setup: benchmarks[k][0]})
```

```
}
```

Language: JavaScript, Filename: benchmark-write-access/benchmark.js

```
'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 11:09:18 am'!
Object subclass: #ConstraintMockObject
  instanceVariableNames: 'a b c d e'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Constraints-Benchmarks'!

"... accessors and initializers for the mock object ..."

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 10:53:31 am'!
Benchmark subclass: #ConstraintsBenchmarks
  instanceVariableNames: 'obj constrObj constraints'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Constraints-Benchmarks'!

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:42'!
benchConstrainedWriteAccess

  1000 timesRepeat: [
    constrObj a: constrObj a + 1.
    constrObj b: constrObj b + 1.
    constrObj c: constrObj c + 1.
    constrObj d: constrObj d + 1.
    constrObj e: constrObj e + 1.
  ]! !

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:47'!
benchConstrainedWriteAccessDisabled

  constraints do: #disable.
  1000 timesRepeat: [
    constrObj a: constrObj a + 1.
    constrObj b: constrObj b + 1.
    constrObj c: constrObj c + 1.
    constrObj d: constrObj d + 1.
    constrObj e: constrObj e + 1.
  ]! !

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:39'!
benchWriteAccess

  1000 timesRepeat: [
    obj a: obj a + 1.
    obj b: obj b + 1.
    obj c: obj c + 1.
    obj d: obj d + 1.
    obj e: obj e + 1.
  ]! !


!ConstraintsBenchmarks methodsFor: 'running' stamp: 'tfel 9/23/2015 15:46'!
setUp

  obj := ConstraintMockObject new.
  constrObj := ConstraintMockObject new.
  constraints := OrderedCollection new.
  constraints add: [constrObj a >= 1] alwaysTrue.
  constraints add: [constrObj b >= 1] alwaysTrue.
  constraints add: [constrObj c >= 1] alwaysTrue.
  constraints add: [constrObj d >= 1] alwaysTrue.
  constraints add: [constrObj e >= 1] alwaysTrue.
! !
```

Language: Squeak Changesets, Filename: benchmark-write-access/benchmark.st

Edit Constraints

The following code listings show how we measured the performance impact of edit constraints compared to purely imperative code and constraint code that does not use edit constraints in Ruby and JavaScript. Squeak/Smalltalk is omitted, because that prototype did not implement support for edit constraints.

```
require "mybenchmark"
require "libcassowary"

class Mercury
  attr_accessor :top, :bottom
  def initialize
    @top = 10
    @bottom = 0
  end

  def height
    @top - @bottom
  end
end

class Mouse
  attr_accessor :location_y
  def initialize
```

```
    @location_y = 10
  end
end

class Rectangle
  attr_accessor :top, :bottom
  def initialize(name, top, bottom)
    @name = name
    @top = top
    @bottom = bottom
  end
end

class Thermometer < Rectangle
  def initialize(top, bottom)
    super("thermometer", top, bottom)
  end
end

class Display
  attr_accessor :number
  def initialize
    @number = 0
  end
end

mouse = mercury = thermometer = grey = white = temperature = display = nil

Resetter = Proc.new do |label|
  if label.start_with? "Imperative"
    mouse = Mouse.new
    mercury = Mercury.new
    thermometer = Thermometer.new(200, 0)
    grey = Rectangle.new("grey", mercury.top, mercury.bottom)
    white = Rectangle.new("white", thermometer.top, mercury.top)
    temperature = mercury.height
    display = Display.new
  end
  if label.start_with? "Declarative" or label.start_with? "Edit"
    mouse = Mouse.new
    mercury = Mercury.new
    thermometer = Thermometer.new(200, 0)
    grey = Rectangle.new("grey", mercury.top, mercury.bottom)
    white = Rectangle.new("white", thermometer.top, mercury.top)
    temperature = mercury.height
    display = Display.new

    always { temperature == mercury.height }
    always { white.top == thermometer.top }
    always { white.bottom == mercury.top }
    always { grey.top == mercury.top }
    always { grey.bottom == mercury.bottom }
    always { display.number == temperature }
    always { mercury.top == mouse.location_y }
    always { mercury.top <= thermometer.top }
    always { mercury.bottom == thermometer.bottom }
  end
end

class SetupSuite
  def warming(label, *args)
    Resetter[label]
  end
  def warmup_stats(*); end
  alias_method :add_report, :warmup_stats
  alias_method :running, :warming
end
suite = SetupSuite.new

Benchmark.ips do |x|
  x.config(:suite => suite)
  x.report('Imperative Drag Simulation') do |t|
    t.times do
      100.times do |i|
        mouse.location_y = i
        old = mercury.top
        mercury.top = mouse.location_y
        if mercury.top > thermometer.top
          mercury.top = thermometer.top
        end
        temperature = mercury.top
        if (old < mercury.top)
          # moves upwards (draws over the white)
          grey.top = mercury.top
        else
          # moves downwards (draws over the grey)
          white.bottom = mercury.top
        end
        display.number = temperature
      end
    end
  end
  x.report('Declarative Drag Simulation') do |t|
    t.times { 100.times { |i| mouse.location_y = i } }
  end
  x.report('Edit Drag Simulation') do |t|
    t.times do
      edit(stream: 100.times.each, accessors: []) { mouse.location_y }
    end
  end
  x.compare!
end
```

Language: Ruby, Filename: benchmark-edit/benchmark.rb

```
suite = new Benchmark.Suite;
```

```
benchmarks = {
   "Imperative Drag": [function() {
      mouse = {location_y: 0},
      mercury = {top: 0, bottom: 0},
      thermometer = {top: 0, bottom: 0},
      temperature = 0,
      gray = {top: 0, bottom: 0},
      white = {top: 0, bottom: 0},
      display = {number: 0};
   }, function() {
      for (var i = 0; i < 100; i++) {
         mouse.location_y = i
         var old = mercury.top
         mercury.top = mouse.location_y
         if (mercury.top > thermometer.top) {
            mercury.top = thermometer.top
         }
         temperature = mercury.top
         if (old < mercury.top) {
            // moves upwards (draws over the white)
            gray.top = mercury.top
         } else {
            // moves downwards (draws over the gray)
            white.bottom = mercury.top
         }
         display.number = temperature
      }
   }],
   "Declarative Drag": [resetConstraints = function() {
      ctx = {
         mouse: {location_y: 0},
         mercury: {top: 0, bottom: 0},
         thermometer: {top: 0, bottom: 0},
         temperature: {c: 0},
         gray: {top: 0, bottom: 0},
         white: {top: 0, bottom: 0},
         display: {number: 0}};
      solver = new ClSimplexSolver();
      bbb.always({solver: solver, ctx: ctx}, function () { return temperature.c == mercury.top });
      bbb.always({solver: solver, ctx: ctx}, function () { return white.top == thermometer.top });
      bbb.always({solver: solver, ctx: ctx}, function () { return white.bottom == mercury.top });
      bbb.always({solver: solver, ctx: ctx}, function () { return gray.top == mercury.top });
      bbb.always({solver: solver, ctx: ctx}, function () { return gray.bottom == mercury.bottom });
      bbb.always({solver: solver, ctx: ctx}, function () { return display.number == temperature.c });
      bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top == mouse.location_y });
      bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top <= thermometer.top });
      bbb.always({solver: solver, ctx: ctx}, function () { return mercury.bottom == thermometer.bottom });
   }, function() {
      for (var i = 0; i < 100; i++) {
         ctx.mouse.location_y = i
      }
   }],
   "Edit Drag": [function() {
      resetConstraints();
      cb = bbb.edit(ctx.mouse, ["location_y"]);
   }, function() {
      for (var i = 0; i < 100; i++) {
         cb([i]);
      }
   }]
}

for (var k in benchmarks) {
   suite.add(k, benchmarks[k][1], {setup: benchmarks[k][0]})
}
```

Language: JavaScript, Filename: benchmark-edit/benchmark.js

Edit Constraint JIT

The following code listing shows the impact of our prototype edit constraint JITs in JavaScript. It shows that the JIT works for the benchmarks shown in Section B, as well as some micro-benchmarks that test specific circumstances like changing two variables at different frequencies.

```
var suite = new Benchmark.Suite;

benchmarks = {
   "Imperative Drag": [function() {
      mouse = {location_y: 0},
      mercury = {top: 0, bottom: 0},
      thermometer = {top: 0, bottom: 0},
      temperature = 0,
      gray = {top: 0, bottom: 0},
      white = {top: 0, bottom: 0},
      display = {number: 0};
   }, function() {
      for (var i = 0; i < 100; i++) {
         mouse.location_y = i
         var old = mercury.top
         mercury.top = mouse.location_y
         if (mercury.top > thermometer.top) {
            mercury.top = thermometer.top
         }
         temperature = mercury.top
         if (old < mercury.top) {
            // moves upwards (draws over the white)
            gray.top = mercury.top
         } else {
            // moves downwards (draws over the gray)
```

```
            white.bottom = mercury.top
        }
        display.number = temperature
    }
}],
"Declarative Drag (Classic JIT)": [constrainf = function(jit) {
    ctx = {
        mouse: {location_y: 0},
        mercury: {top: 0, bottom: 0},
        thermometer: {top: 0, bottom: 0},
        temperature: {c: 0},
        gray: {top: 0, bottom: 0},
        white: {top: 0, bottom: 0},
        display: {number: 0}};
    solver = new ClSimplexSolver();
    solver.ecjit = jit || new ClassicECJIT();
    bbb.always({solver: solver, ctx: ctx}, function () { return temperature.c == mercury.top });
    bbb.always({solver: solver, ctx: ctx}, function () { return white.top == thermometer.top });
    bbb.always({solver: solver, ctx: ctx}, function () { return white.bottom == mercury.top });
    bbb.always({solver: solver, ctx: ctx}, function () { return gray.top == mercury.top });
    bbb.always({solver: solver, ctx: ctx}, function () { return gray.bottom == mercury.bottom });
    bbb.always({solver: solver, ctx: ctx}, function () { return display.number == temperature.c });
    bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top == mouse.location_y });
    bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top <= thermometer.top });
    bbb.always({solver: solver, ctx: ctx}, function () { return mercury.bottom == thermometer.bottom });
}, benchf = function() {
    for (var i = 0; i < 100; i++) {
        ctx.mouse.location_y = i
    }
}],
"Declarative Drag (Additive JIT)": [
    function() { constrainf(new AdditiveAdaptiveECJIT()) },
    benchf
],
"Declarative Drag (Multiplicative JIT)": [
    function() { constrainf(new MultiplicativeAdaptiveECJIT()) },
    benchf
],
"Declarative Drag (Last JIT)": [
    function() { constrainf(new LastECJIT()) },
    benchf
],
}

for (var k in benchmarks) {
    // suite.add(k, benchmarks[k][1], {setup: benchmarks[k][0]})
}

ecbenchmarks1 = {
    'dbAddSim': [
        function(ecjit) {
            o = {x: 0, y: 0, z: 0};
            solver = new DBPlanner();
            solver.ecjit = ecjit;

            bbb.always({solver: solver, ctx: {o: o}}, function () {
                return o.x == o.z - o.y &&
                    o.y == o.z - o.x &&
                    o.z == o.x + o.y;
            })
        }, function(iterations) {
            for (var i = 0; i < iterations; i++) {
                o.x = i;
                console.assert(o.x + o.y == o.z);
            }
        }, function(iterations) {
            cb = bbb.edit(o, ["x"]);
            for (var i = 0; i < iterations; i++) {
                cb([i]);
                console.assert(o.x + o.y == o.z);
            }
            cb();
        }],
    'clAddSim': [
        function(ecjit) {
            o = {x: 0, y: 0, z: 0};
            solver = new ClSimplexSolver();
            solver.ecjit = ecjit;
            solver.setAutosolve(false);
            bbb.always({solver: solver, ctx: {o: o}}, function () { return o.x + o.y == o.z; });
        }, function(iterations) {
            for (var i = 0; i < iterations; i++) {
                o.x = i;
                console.assert(o.x + o.y == o.z);
            }
        }, function() {
            cb = bbb.edit(o, ["x"]);
        }, function (iterations) {
            for (var i = 0; i < iterations; i++) {
                cb([i]);
                console.assert(o.x + o.y == o.z);
            }
        }],
    'clDragSim': [
        function(ecjit) {
            ctx = {
                mouse: {location_y: 0},
                mercury: {top: 0, bottom: 0},
                thermometer: {top: 0, bottom: 0},
                temperature: {c: 0},
                gray: {top: 0, bottom: 0},
                white: {top: 0, bottom: 0},
                display: {number: 0}},
            solver = new ClSimplexSolver();
            solver.ecjit = ecjit;
            solver.setAutosolve(false);

            bbb.always({solver: solver, ctx: ctx}, function () { return temperature.c == mercury.top });
            bbb.always({solver: solver, ctx: ctx}, function () { return white.top == thermometer.top });
            bbb.always({solver: solver, ctx: ctx}, function () { return white.bottom == mercury.top });
```

```
         bbb.always({solver: solver, ctx: ctx}, function () { return gray.top == mercury.top });
         bbb.always({solver: solver, ctx: ctx}, function () { return gray.bottom == mercury.bottom });
         bbb.always({solver: solver, ctx: ctx}, function () { return display.number == temperature.c });
         bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top == mouse.location_y });
         bbb.always({solver: solver, ctx: ctx}, function () { return mercury.top <= thermometer.top });
         bbb.always({solver: solver, ctx: ctx}, function () { return mercury.bottom == thermometer.bottom });
      }, function(iterations) {
         for (var i = 0; i < iterations; i++) {
            ctx.mouse.location_y = i;
            console.assert(ctx.mouse.location_y == i);
         }
      }, function() {
         cb = bbb.edit(ctx.mouse, ["location_y"]);
      }, function(iterations) {
         for (var i = 0; i < iterations; i++) {
            cb([i]);
            console.assert(ctx.mouse.location_y == i);
         }
      }],
}

jits = ['EmptyECJIT', 'ClassicECJIT', 'AdditiveAdaptiveECJIT', 'MultiplicativeAdaptiveECJIT', 'LastECJIT']
iterations = 1;

var idx = 0
function wrap(options, fn) {
   var str = fn.toString();
   for (var k in options) {
      window[k + idx] = options[k];
      str = str.replace(RegExp("([^a-zA-Z])" + k + "([^a-zA-Z0-9])", "g"), "$1" + k + idx + "$2");
      idx += 1;
   }
   return eval("(" + str + ")");
}

for (var k in ecbenchmarks1) {
   jits.each(function (jit) {
      ecjit = eval(jit)
      suite.add(k + " " + jit, wrap({k: k, iterations: iterations}, function() {
         ecbenchmarks1[k][1](iterations)
      }), {setup: wrap({k: k, ecjit: ecjit}, function() {
         ecbenchmarks1[k][0](new ecjit)
      })})
   });
   suite.add(k + " Edit Constraints", wrap({k: k, iterations: iterations}, function() {
      ecbenchmarks1[k][3](iterations)
   }), {setup: wrap({k: k, ecjit: ecjit}, function() {
      ecbenchmarks1[k][2]()
   })})
}

[1, 3].each(function(sheer) {
   jits.each(function(jit) {
      var ecjit = new (eval(jit))
      var numIterations = iterations;
      var setup = function() {
         ctx = {
            mouse: {x: 100, y: 100},
            wnd: {w: 100, h: 100},
            comp1: {w: 70, display: 0},
            comp2: {w: 30, display: 0}
         };
         var solver = new ClSimplexSolver();
         solver.ecjit = ecjit;
         solver.setAutosolve(false);

         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.w == mouse.x });
         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.h == mouse.y });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w <= 400; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w+comp2.w == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.display == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp2.display == wnd.h; });
      };
      suite.add('clDrag2DSim' + sheer + " " + jit,
             wrap({sheer: sheer, numIterations: numIterations, ecjit: ecjit}, function() {
                for(var i = 0; i < numIterations; i++) {
                   ctx.mouse.x = 100+i;
                   if(i % sheer == 0) {
                      ctx.mouse.y = 100+i;
                   }
                   console.assert(ctx.mouse.x == 100+i);
                   if(i % sheer == 0) {
                      console.assert(ctx.mouse.y == 100+i);
                   }
                }
             }),
             {setup: wrap({sheer: sheer, numIterations: numIterations, ecjit: ecjit}, setup)});
      suite.add('clDrag2DSimEdit' + sheer + " " + jit,
             wrap({sheer: sheer, numIterations: numIterations, ecjit: ecjit}, function(numIterations, sheer) {
                for(var i = 0; i < numIterations; i++) {
                   cb([100+i, Math.floor((100+i)/sheer)*sheer]);
                   console.assert(ctx.mouse.x == 100+i);
                   console.assert(ctx.mouse.y == Math.floor((100+i)/sheer)*sheer);
                }
             }),
             {setup: wrap({sheer: sheer, numIterations: numIterations, ecjit: ecjit}, function() {
                setup();
                cb = bbb.edit(ctx.mouse, ["x", "y"]);
             })});
   });
});

[iterations / 2, iterations / 10].each(function(numSwitch) {
   jits.each(function(jit) {
      var ecjit = new (eval(jit))
      var numIterations = iterations;
      var setup = wrap({numSwitch: numSwitch, numIterations: numIterations, ecjit: ecjit}, function() {
         ctx = {
            mouse: {x: 100, y: 100},
            wnd: {w: 100, h: 100},
```

```
            comp1: {w: 70, display: 0},
            comp2: {w: 30, display: 0}
         };
         var solver = new ClSimplexSolver();
         solver.ecjit = ecjit;
         solver.setAutosolve(false);

         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.w == mouse.x });
         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.h == mouse.y });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w <= 400; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w+comp2.w == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.display == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp2.display == wnd.h; });
      });
      suite.add('clDrag2DSimChange' + numSwitch + " " + jit,
             wrap({numSwitch: numSwitch, numIterations: numIterations, ecjit: ecjit}, function() {
                for(var i = 0; i < numIterations; i++) {
                   if(i < numSwitch) {
                      ctx.mouse.x = 100+i;
                      console.assert(ctx.mouse.x == 100+i);
                   } else {
                      ctx.mouse.y = 100+(i-numSwitch);
                      console.assert(ctx.mouse.x == numSwitch-1);
                      console.assert(ctx.mouse.y == 100+(i-numSwitch));
                   }
                }
             }),
             {setup: setup});
      suite.add('clDrag2DSimChangeEdit' + numSwitch + " " + jit,
             wrap({numSwitch: numSwitch, numIterations: numIterations, ecjit: ecjit}, function() {
                for(var i = 0; i < numIterations; i++) {
                   if(i < numSwitch) {
                      cb([100+i]);
                      console.assert(ctx.mouse.x == 100+i);
                   } else {
                      if(i == numSwitch) {
                         cb();
                         cb = bbb.edit(ctx.mouse, ["y"]);
                      }
                      cb([100+(i-numSwitch)]);
                      console.assert(ctx.mouse.x == numSwitch-1);
                      console.assert(ctx.mouse.y == 100+(i-numSwitch));
                   }
                }
             }),
             {setup: function() {
                setup();
                cb = bbb.edit(ctx.mouse, ["x"]);
             }});
   });
});

[5, 10].each(function(switchFreq) {
   jits.each(function(jit) {
      var ecjit = new (eval(jit))
      var numIterations = iterations;
      var setup = wrap({switchFreq: switchFreq, numIterations: numIterations, ecjit: ecjit}, function() {
         ctx = {
            mouse: {x: 100, y: 100},
            wnd: {w: 100, h: 100},
            comp1: {w: 70, display: 0},
            comp2: {w: 30, display: 0}
         };
         var solver = new ClSimplexSolver();
         solver.ecjit = ecjit;
         solver.setAutosolve(false);

         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.w == mouse.x });
         bbb.always({solver: solver, ctx: ctx}, function () { return wnd.h == mouse.y });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w <= 400; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.w+comp2.w == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp1.display == wnd.w; });
         bbb.always({solver: solver, ctx: ctx}, function () { return comp2.display == wnd.h; });
      });
      suite.add('clDrag2DSimFreqChange' + switchFreq + " " + jit,
             wrap({switchFreq: switchFreq, numIterations: numIterations, ecjit: ecjit}, function() {
                for(var i = 0; i < numIterations; i++) {
                   if(i % (switchFreq*2) < switchFreq) {
                      ctx.mouse.x = 100+i;
                      console.assert(ctx.mouse.x == 100+i);
                   } else {
                      ctx.mouse.y = 100+i;
                      console.assert(ctx.mouse.y == 100+i);
                   }
                }

             }),
             {setup: setup});
      suite.add('clDrag2DSimChangeEdit' + switchFreq + " " + jit,
             wrap({switchFreq: switchFreq, numIterations: numIterations, ecjit: ecjit}, function() {
                for(var i = 0; i < numIterations; i++) {
                   if(i % (switchFreq*2) < switchFreq) {
                      cb([100+i, 100+i/(switchFreq*2)]);
                      console.assert(ctx.mouse.x == 100+i);
                   } else {
                      cb([100+i/(switchFreq*2), 100+i]);
                      console.assert(ctx.mouse.y == 100+i);
                   }
                }
             }),
             {setup: function() {
                setup();
                cb = bbb.edit(ctx.mouse, ["x", "y"]);
             }});
   });
});
```

Language: JavaScript, Filename: benchmark-jit/jitbenchmarks.js

Constraint-* Language Comparisons

In the following subsections, we show the code listings for the three cross-language constraint benchmarks that we ran to compare Babelsberg performance to Prolog, Kaplan, and Turtle. Each subsection presents the code in Babelsberg/R in Ruby, Babelsberg/JS in JavaScript, Babelsberg/S in Squeak/Smalltalk, Prolog using the constraint-logic programming library, Kaplan, and Turtle. Depending on the language, repetitions are sometimes hard coded and at other times were injected by an outside benchmark runner. Each of these benchmarks were run 1, 3, 5, 10, 50, and 100 times, either by configuring the benchmark runner or tweaking the hard coded repetitions.

Animals Puzzle

```ruby
require "libz3"

CENTS = 10000
ANIMALS = 100
DOGC = 1500
CATC = 100
MICEC = 25

def action
  cents, animals, dogc, catc, micec = [0.0] * 5

  always { cents == CENTS &&
        animals == ANIMALS &&
        dogc == DOGC &&
        catc == CATC &&
        micec == MICEC }

  dog, cat, mouse = 0, 0, 0
  c1 = always { dog >= 1 && cat >= 1 && mouse >= 1 }
  c2 = always { dog + cat + mouse == animals }
  c3 = always {dog * dogc + cat * catc + mouse * micec == cents }
  puts "Dogs: #{dog}, cats: #{cat}, mice: #{mouse}"
  [c1,c2,c3].each(&:disable)
end
```

Language: Ruby, Filename: benchmark-constraint-languages/animals.rb

```javascript
CENTS = 10000
ANIMALS = 100
DOGC = 1500
CATC = 100
MICEC = 25
TIME_ = 0
REPEATS = 10

function action() {
   var solver = new EmZ3();
   var oldPM = solver.postMessage;
   solver.postMessage = function(string) {
      return oldPM.apply(solver, [string.replace(/Real/g, "Int")]);
   }

   setTimeout(function () {
      var start = Date.now();
      var obj = {cents: 0, animals: 0, dogc: 0, catc: 0, micec: 0};
      bbb.always({
         solver: solver,
         ctx: {obj: obj}}, function () {
            return obj.cents == CENTS &&
               obj.animals == ANIMALS &&
               obj.dogc == DOGC &&
               obj.catc == CATC &&
               obj.micec == MICEC });

      var obj2 = {dog: 0, cat: 0, mouse: 0};
      bbb.always({
         solver: solver,
         ctx: {obj2: obj2}}, function () { return obj2.dog >= 1 && obj2.cat >= 1 && obj2.mouse >= 1 });
      bbb.always({
         solver: solver,
         ctx: {obj: obj, obj2: obj2}}, function () { return obj2.dog + obj2.cat + obj2.mouse == obj.animals });
      bbb.always({
         solver: solver,
         ctx: {obj: obj, obj2: obj2}}, function () {
           return obj2.dog * obj.dogc + obj2.cat * obj.catc + obj2.mouse * obj.micec == obj.cents
         });
      console.log("Dogs: " + obj2.dog + ", cats: " + obj2.cat + ", mice: " + obj2.mouse)
      TIME_ += (Date.now() - start);

      if (REPEATS > 0) {
         REPEATS -= 1;
         setTimeout(action, 0);
      } else {
         console.log("THIS IS THE TIME:" + TIME_);
         alert.original.apply(window, ["CLOSE ME"])
      }
   }, 3000);
}
```

Language: JavaScript, Filename: benchmark-constraint-languages/animals.js

```
'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 5 October 2015 at 11:56:17 am'!
!ConstraintZ3Variable methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:33'!
initialize
   "Use integers"
   varName := 'noName'.
   type := 'Int'.! !

Object subclass: #AnimalsObject
      instanceVariableNames: 'cents animals dogc catc micec dog cat mouse'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Constraints-Benchmarks'!

"... accessors and initializers for the animals object ..."

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 5 October 2015 at 12:01:34 pm'!
Benchmark subclass: #ConstraintsBenchmarks
      instanceVariableNames: 'obj'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Constraints-Benchmarks'!

!ConstraintsBenchmarks methodsFor: 'nil' stamp: 'tfel 10/5/2015 12:01'!
benchAnimals
10 timesRepeat: [
      | solver |
      obj := AnimalsObject new.
      solver := ConstraintSolver newZ3Solver.
      [ (obj cents = 10000) &
            (obj animals = 100) & (
                  obj dogc = 1500) & (
                        obj catc = 100) & (
                              obj micec = 25) ] alwaysSolveWith: solver.
      [ obj dog >= 1 & ( obj cat >= 1 ) & ( obj mouse >= 1) ] alwaysSolveWith: solver.
      [ obj dog + obj cat + obj mouse = obj animals ] alwaysSolveWith: solver.
      [ (obj dog * obj dogc) + (obj cat * obj catc) + (obj mouse * obj micec) = obj cents ] alwaysSolveWith: solver.
      FileStream stdout nextPutAll: 'Dogs: ', obj dog, ', cats: ', obj cat , ', mice: ', obj mouse; cr; flush].
! !
```

Language: Squeak Changesets, Filename: benchmark-constraint-languages/animals.st

```
:- use_module(library(clpfd)).

animals(Vars) :-
   Animals is 100,
   Cents is 10000,
   Dogc is 1500,
   Catc is 100,
   Micec is 25,
   Vars = [Dog,Cat,Mouse],
   Vars ins 1..Animals,
   Dogc*Dog + Catc*Cat + Micec*Mouse #= Cents,
   Dog + Cat + Mouse #= Animals.

bench(Count) :-
   T1 is cputime,
   dobench(Count),
   T2 is cputime,
   report(Count, T1, T2).

dobench(Count) :-
   nrepeat(Count),
   Vars = [Dog,Cat,Mouse],
   animals(Vars),
   label(Vars),
   write(Vars),
   nl,
   fail.
dobench(_).

nrepeat(_).
nrepeat(N) :-
  N>1,
  N1 is N-1,
  nrepeat(N1).

report(Count, T1, T2) :-
   Time is T2-T1,
   nl,
   write('THIS IS THE TIME: '),
   write(Time),
   nl.
```

Language: Prolog, Filename: benchmark-constraint-languages/animals.prolog

```
import z3.scala._
import cp.Definitions._
import cp.Terms._
import cp.LTrees._
import cp.ConstraintSolving
import purescala.FairZ3Solver

object Animals extends App {
  object Example {
    val name = "Animals"

    def run : Unit = {
      println("*** Running " + name + " ***")
      action
    }

    def asserting(c : Constraint0) : Unit = {
      var entered = false
```

```
    for(i <- c.lazyFindAll) {
      entered = true
    }
    if(!entered) { throw new Exception("Asserting failed.") }
   }

   def action : Unit = {
    val anyInt : Constraint1[Int] = ((n : Int) => true)

    val cents = anyInt.lazySolve
    val animals = anyInt.lazySolve
    val dogc = anyInt.lazySolve
    val catc = anyInt.lazySolve
    val micec = anyInt.lazySolve

    asserting( cents == 10000 )
    asserting( animals == 100 )
    asserting( dogc == 1500 )
    asserting( catc == 100 )
    asserting( micec == 25 )

    val dog = anyInt.lazySolve
    val cat = anyInt.lazySolve
    val mouse = anyInt.lazySolve

    asserting( dog >= 1 )
    asserting( cat >= 1 )
    asserting( mouse >= 1 )
    asserting( animals == dog + cat + mouse )
    asserting( cents == dog * dogc + cat * catc + mouse * micec )

    println("Dogs: " + dog.value + ", cats: " + cat.value + ", mice: " + mouse.value)
   }
  }
}
```

Language: Kaplan, Filename: benchmark-constraint-languages/animals.kaplan

```
module animals;

import io, sys.times;

fun test0()
  var cents: !int := var 0;
  var animals: !int := var 0;
  var dogc: !int := var 0;
  var catc: !int := var 0;
  var micec: !int := var 0;

  require cents = 10000;
  require animals = 100;
  require dogc = 1500;
  require catc = 100;
  require micec = 25;

  var dog: !int := var 0;
  var cat: !int := var 0;
  var mouse: !int := var 0;

  require dog >= 1;
  require cat >= 1;
  require mouse >= 1;
  require dog < animals;
  require cat < animals;
  require mouse < animals;
  require dog + cat + mouse = animals;
  require dog * dogc + cat * catc + mouse * micec = cents;

  io.put ("Dogs: "); io.put (!dog);
  io.put (", cats: "); io.put (!cat);
  io.put (", mice: "); io.put (!mouse); io.nl ();
end;
```

Language: Turtle, Filename: benchmark-constraint-languages/animals.turtle

Layout Constraints

```
require "libz3"

def fun
  gap, pw, lw, rw = [0]*4

  c1 = always { pw == 40000 }
  c2 = always { gap == pw / 20000 }
  c3 = always { lw + gap + rw == pw }
  c4 = always { lw >= 0 }
  c5 = always { rw >= 0 }

  puts "gap #{gap}, left column #{lw}, right column #{rw}, page width #{pw}"
  [c1,c2,c3,c4,c5].each(&:disable)
end
```

Language: Ruby, Filename: benchmark-constraint-languages/layout.rb

```
function fun() {
   var solver = new ClSimplexSolver();
   var obj = {gap: 0,
              pw: 0,
```

```
          lw: 0,
          rw: 0};
   bbb.always({solver: solver, ctx: {obj: obj}}, function () { return obj.pw == 40000 });
   bbb.always({solver: solver, ctx: {obj: obj}}, function () { return obj.gap == obj.pw / 20000 });
   bbb.always({solver: solver, ctx: {obj: obj}}, function () { return obj.lw + obj.gap + obj.rw == obj.pw });
   bbb.always({solver: solver, ctx: {obj: obj}}, function () { return obj.lw >= 0 });
   bbb.always({solver: solver, ctx: {obj: obj}}, function () { return obj.rw >= 0 });
   console.log("gap " + obj.gap + ", left column " + obj.lw + ", right column " + obj.rw + ", page width " + obj.pw)
}
```

Language: JavaScript, Filename: benchmark-constraint-languages/layout.js

```
'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 11:09:18 am'!
Object subclass: #ConstraintMockObject
      instanceVariableNames: 'gap pw lw rw'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Constraints-Benchmarks'!

"... accessors and initializers for the mock object ..."

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 30 September 2015 at 10:53:31 am'!
Benchmark subclass: #ConstraintsBenchmarks
      instanceVariableNames: 'obj constrObj constraints'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Constraints-Benchmarks'!

!ConstraintsBenchmarks methodsFor: 'benchmarks' stamp: 'tfel 9/23/2015 15:42'!
benchLayout

  10 timesRepeat: [
    obj := ConstraintMockObject new.
    [ obj pw = 40000 ] alwaysTrue.
    [ obj gap = (obj pw / 20000) ] alwaysTrue.
    [ obj lw + obj gap + obj rw = obj pw ] alwaysTrue.
    [ obj lw >= 0 ] alwaysTrue.
    [ obj rw >= 0 ] alwaysTrue.
    FileStream stdout
      nextPutAll: 'gap = ', obj gap, ', pw = ', obj pw, ' lw = ', obj lw, ' rw = ', obj rw;
      crlf; flush.
  ]
! !
```

Language: Squeak Changesets, Filename: benchmark-constraint-languages/layout.st

```
:- use_module(library(clpfd)).

layout(Vars) :-
   Vars = [Gap,Pw,Lw,Rw],
   Pw #= 40000,
   Gap * 20000 #= Pw,
   Lw + Gap + Rw #= Pw,
   Lw #>= 0,
   Rw #>= 0,
   Gap #= 2,
   Lw #= 0.

bench(Count) :-
   T1 is cputime * 1000,
   dobench(Count),
   T2 is cputime * 1000,
   report(Count, T1, T2).

dobench(Count) :-
   nrepeat(Count),
   Vars = [Gap,Pw,Lw,Rw],
   layout(Vars),
   label(Vars),
   write(Vars),
   nl,
   fail.
dobench(_).

nrepeat(_).
nrepeat(N) :-
  N>1,
  N1 is N-1,
  nrepeat(N1).

report(Count, T1, T2) :-
   Time is T2-T1,
   nl,
   write('THIS IS THE TIME: '),
   write(Time),
   nl.
```

Language: Prolog, Filename: benchmark-constraint-languages/layout.prolog

```
import z3.scala._
import cp.Definitions._
import cp.Terms._
import cp.LTrees._
import cp.ConstraintSolving
import purescala.FairZ3Solver

object Layout extends App {
  object Example {
    val name = "Turtle Layout"

    def run : Unit = {
```

```
    println("*** Running " + name + " ***")
    action
  }

  def asserting(c : Constraint0) : Unit = {
    var entered = false
    for(i <- c.lazyFindAll) {
     entered = true
    }
    if(!entered) { throw new Exception("Asserting failed.") }
  }

  def action : Unit = {
    val anyReal : Constraint1[Int] = ((n : Int) => true)

    val gap = anyReal.lazySolve
    val pw = anyReal.lazySolve
    val rw = anyReal.lazySolve
    val lw = anyReal.lazySolve

    asserting( pw == 40000 )
    asserting( gap == pw / 20000 )
    asserting( pw == lw + gap + rw )
    asserting( lw >= 0 )
    asserting( rw >= 0 )

    println("gap " + gap.value + ", left column " + lw.value + ", right column " + rw.value + ", page width " + pw.value)
  }
 }
}
```

Language: Kaplan, Filename: benchmark-constraint-languages/layout.kaplan

```
// layout.t -- A simple layout example.
// Copyright (C) 2003 Martin Grabmueller <mgrabmue@cs.tu-berlin.de>
module layout;

import io, sys.times;

fun test0()
 // Gap between the columns.
 var gap: !real := var 0.0;
 // Page width.
 var pw: !real := var 0.0;
 // Left and right column.
 var lw: !real := var 0.0;
 var rw: !real := var 0.0;

 require pw = 40000.0;
 require gap = (pw / 20000.0);
 require lw + gap + rw = pw;
 require lw >= 0.0;
 require rw >= 0.0;

 io.put ("gap = "); io.put (!gap); io.nl ();
 io.put ("lw = "); io.put (!lw); io.nl ();
 io.put ("rw = "); io.put (!rw); io.nl ();
 io.put ("pw = "); io.put (!pw); io.nl ();
end;
```

Language: Turtle, Filename: benchmark-constraint-languages/layout.turtle

Send+More=Money Puzzle

```
require "libz3"
require "libarraysolver"

class Array
 def ins(range)
   return true if self.empty?
   self[1..-1].ins(range) &&
    self[0] >= range.first &&
    self[0] <= range.last
 end
end

def action
 s,e,n,d,m,o,r,y = [0]*8

 # each digit is between 0 and 9
 c = always { [s,e,n,d,m,o,r,y].ins(0..9) }

 c1 = always { [s,e,n,d,m,o,r,y].alldifferent? }

 c2 = always do
   s*1000 + e*100 + n*10 + d +
   m*1000 + o*100 + r*10 + e ==
   m*10000 + o*1000 + n*100 + e*10 + y
 end

 c3 = always { s>0 && m>0 }

 puts ("solution: [s,e,n,d,m,o,r,y] = " + [s,e,n,d,m,o,r,y].to_s)
 [c, c1, c2, c3].each(&:disable)
end
```

Language: Ruby, Filename: benchmark-constraint-languages/sendmoremoney.rb

```
CENTS = 10000
ANIMALS = 100
DOGC = 1500
CATC = 100
MICEC = 25
TIME_ = 0
REPEATS = 10

function action() {
   var solver = new EmZ3();
   var oldPM = solver.postMessage;
   solver.postMessage = function(string) {
      return oldPM.apply(solver, [string.replace(/Real/g, "Int")]);
   }

   setTimeout(function () {
      var start = Date.now();
      var obj = {s: 0, e: 0, n: 0, d: 0, m: 0, o: 0, r: 0, y:0};
      var names = ["s", "e", "n", "d", "m", "o", "r", "y"];
      var fo;
      names.forEach(function (f) {
         bbb.always({
            solver: solver,
            ctx: {obj: obj, f: f}}, function () {
               return obj[f] <= 9 && obj[f] >= 0 });
      });
      bbb.always({
         solver: solver,
         ctx: {obj: obj}}, function () {
            return obj.s != 0 && obj.m != 0 });
      names.forEach(function (f) {
         names.forEach(function (fo) {
            if (f !== fo) {
               bbb.always({
                  solver: solver,
                  ctx: {obj: obj, f: f, fo: fo}}, function () {
                     return obj[f] != obj[fo] });
            }
         })
      })

      bbb.always({
         solver: solver,
         ctx: {obj: obj}}, function () {
            return obj.s * 1000 + obj.e * 100 + obj.n * 10 + obj.d +
               obj.m * 1000 + obj.o * 100 + obj.r * 10 + obj.e ==
               obj.m * 10000 + obj.o * 1000 + obj.n * 100 + obj.e * 10 + obj.y });

      console.log("solution: [s,e,n,d,m,o,r,y] = " + [obj.s,obj.e,obj.n,obj.d,obj.m,obj.o,obj.r,obj.y])
      TIME_ += (Date.now() - start);

      if (REPEATS > 0) {
         REPEATS -= 1;
         setTimeout(action, 0);
      } else {
         console.log("THIS IS THE TIME:" + TIME_);
         alert.original.apply(window, ["CLOSE ME"])
      }
   }, 3000);
}
```

Language: JavaScript, Filename: benchmark-constraint-languages/sendmoremoney.js

```
'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 5 October 2015 at 11:56:17 am'!
!ConstraintZ3Variable methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:33'!
initialize
   "Use integers"
   varName := 'noName'.
   type := 'Int'.! !

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 5 October 2015 at 1:48:46 pm'!
Object subclass: #SMM
  instanceVariableNames: 'ary'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Constraints-Benchmarks'!

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
initialize

  ary := {0. 0. 0. 0. 0. 0. 0. 0}! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
d

  ↑ary at: 4! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
e

  ↑ary at: 2! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
m

  ↑ary at: 5! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
n

  ↑ary at: 3! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
o

  ↑ary at: 6! !
```

```
!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
r

 ↑ary at: 7! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
s

 ↑ary at: 1! !

!SMM methodsFor: 'initialize-release' stamp: 'tfel 10/5/2015 13:45'!
y

 ↑ary at: 8! !

!SMM methodsFor: 'as yet unclassified' stamp: 'tfel 10/5/2015 13:46'!
ary

 ↑ary! !

!SMM methodsFor: 'as yet unclassified' stamp: 'tfel 10/5/2015 13:46'!
ary: anArray

 ary := anArray.! !

'From Squeak4.5 of 19 February 2014 [latest update: #13680] on 5 October 2015 at 1:48:48 pm'!
Benchmark subclass: #ConstraintBenchmarks
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Constraints-Benchmarks'!

!ConstraintBenchmarks methodsFor: 'as yet unclassified' stamp: 'tfel 10/5/2015 13:48'!
benchSendMoreMoney
 | solver p |
 50 timesRepeat: [
  p := SMM new.
  solver := ConstraintSolver newZ3Solver.
  [p ary allSatisfy: [:ea | (ea <= 9) & (ea >= 0)]] alwaysSolveWith: solver.
  [p ary allDifferent] alwaysSolveWith: solver.
  [  ((p s * 1000) + (p e * 100) + (p n * 10) + (p d) +
     (p m * 1000) + (p o * 100) + (p r * 10) + (p e)) =
  ((p m * 10000) + (p o * 1000) + (p n * 100) + (p e * 10) + (p y))
  ] alwaysSolveWith: solver.
  [(p s > 0) & (p m > 0)] alwaysSolveWith: solver.
  FileStream stdout nextPutAll: 'solution: ', p ary; cr; flush.
 ]
! !
```

Language: Squeak Changesets, Filename: benchmark-constraint-languages/sendmoremoney.st

```
:- use_module(library(clpfd)).

sendmoremoney(Vars) :-
   Vars = [S,E,N,D,M,O,R,Y],
   Vars ins 0..9,
   S #\= 0,
   M #\= 0,
   all_different(Vars),
            1000*S + 100*E + 10*N + D
   + 1000*M + 100*O + 10*R + E
   #= 10000*M + 1000*O + 100*N + 10*E + Y.

bench(Count) :-
   T1 is cputime,
   dobench(Count),
   T2 is cputime,
   report(Count, T1, T2).

dobench(Count) :-
   nrepeat(Count),
   Vars = [S,E,N,D,M,O,R,Y],
   sendmoremoney(Vars),
   label(Vars),
   write(Vars),
   nl,
   fail.
dobench(_).

nrepeat(_).
nrepeat(N) :-
  N>1,
  N1 is N-1,
  nrepeat(N1).

report(Count, T1, T2) :-
   Time is T2-T1,
   nl,
   write('THIS IS THE TIME: '),
   write(Time),
   nl.
```

Language: Prolog, Filename: benchmark-constraint-languages/sendmoremoney.prolog

```
import z3.scala._
import cp.Definitions._
import cp.Terms._
import cp.LTrees._
import cp.ConstraintSolving
import purescala.FairZ3Solver

object Sendmoremoney extends App {
 object Example {
   val name = "SEND+MORE=MONEY"
```

```
    def run : Unit = {
      println("*** Running " + name + " ***")
      action
    }

    def asserting(c : Constraint0) : Unit = {
      var entered = false
      for(i <- c.lazyFindAll) {
        entered = true
      }
      if(!entered) { throw new Exception("Asserting failed.") }
    }

    def action : Unit = {
      val anyInt : Constraint1[Int] = ((n : Int) => true)

      val letters @ Seq(s,e,n,d,m,o,r,y) = Seq.fill(8)(anyInt.lazySolve)

      for(l <- letters) {
        asserting(l >= 0 && l <= 9)
      }

      when(distinct[Int](s,e,n,d,m,o,r,y)) {
        println("Letters now have distinct values.")
      } otherwise {
        println("Letters can't have distinct values.")
      }

      val fstLine = anyInt.lazySolve
      val sndLine = anyInt.lazySolve
      val total = anyInt.lazySolve

      asserting(fstLine == 1000*s + 100*e + 10*n + d)
      asserting(sndLine == 1000*m + 100*o + 10*r + e)
      asserting(total == 10000*m + 1000*o + 100*n + 10*e + y)

      asserting(s >= 1)
      asserting(m >= 1)

      println("Solution: " + letters.map(_.value) + " (" + fstLine.value + " + " + sndLine.value + " = " + total.value + ")")
    }
  }
}
```

Language: Kaplan, Filename: benchmark-constraint-languages/sendmoremoney.kaplan

```
// sendmory2.t -- Demonstration for Constraint Programming in Turtle.
// Copyright (C) 2003 Martin Grabmueller <mgrabmue@cs.tu-berlin.de>

module sendmory2;

import io, sys.times;

constraint all_different (l: list of !int)
  while tl l <> null do
    var ll: list of !int := tl l;
    while ll <> null do
      require hd l <> hd ll;
      ll := tl ll;
    end;
    l := tl l;
  end;
end;

constraint domain (v: !int, min: int, max: int)
  require v >= min and v <= max;
end;

fun test0()
  var s: !int := var 0;
  var e: !int := var 0;
  var n: !int := var 0;
  var d: !int := var 0;
  var m: !int := var 0;
  var o: !int := var 0;
  var r: !int := var 0;
  var y: !int := var 0;

  require domain (s, 0, 9) and domain (e, 0, 9) and domain (n, 0, 9) and
    domain (d, 0, 9) and domain (m, 1, 9) and domain (o, 0, 9) and
    domain (r, 0, 9) and domain (y, 0, 9) and
    all_different ([s, e, n, d, m, o, r, y]) and
    (s * 1000 + e * 100 + n * 10 + d) +
    (m * 1000 + o * 100 + r * 10 + e) =
    (m * 10000 + o * 1000 + n * 100 + e * 10 + y)
  in
    io.put ("s = "); io.put (!s); io.nl ();
    io.put ("e = "); io.put (!e); io.nl ();
    io.put ("n = "); io.put (!n); io.nl ();
    io.put ("d = "); io.put (!d); io.nl ();
    io.put ("m = "); io.put (!m); io.nl ();
    io.put ("o = "); io.put (!o); io.nl ();
    io.put ("r = "); io.put (!r); io.nl ();
    io.put ("y = "); io.put (!y); io.nl ();
  end;
end;
```

Language: Turtle, Filename: benchmark-constraint-languages/sendmoremoney.turtle

Appendix C.

Constraint Hierarchies

This is a reproduction of the theory of constraint hierarchy relevant to this work, originally published in [33] and given here for convenience.

⁂

Formally, a constraint is a relation over some domain $\mathcal{D}$. The domain $\mathcal{D}$ determines the constraint predicate symbols $\Pi_{\mathcal{D}}$ of the language, so that a constraint is an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in $\Pi_{\mathcal{D}}$ and each t_i is a term.

A *labeled constraint* is a constraint labeled with a priority, written pc, where p is a priority and c is a constraint. For clarity in writing labeled constraints, we give symbolic names to the different priorities. We then map each of these names onto the integers $0 \ldots n$, where n is the number of non-required priorities. Priority 0, with the symbolic name `required`, is always reserved for required constraints.

A constraint system is a multiset H of labeled constraints. Let H_0 denote the required constraints in H, with their labels removed. In the same way, we define the sets $H_1, H_2, \ldots, H_n$ for levels $1, 2, \ldots, n$. We also define $H_k = \emptyset$ for $k > n$.

A *solution* to a set of labeled constraints H is a valuation for the free variables in H, i.e., a function that maps the free variables in H to elements in the domain $\mathcal{D}$. We wish to define the set S of all solutions to H. Clearly, each valuation in S must be such that, after it is applied, all the required constraints hold. In addition, we desire each valuation in S to be such that it satisfies the non-required constraints as well as possible, respecting their relative strengths. To formalize this desire, we first define the set S_0 of valuations such that all the H_0 constraints hold. Then, using S_0, we define the desired set S by eliminating all potential valuations that are worse than some other potential valuation using the comparator predicate *better*. (In the definition, $c\vartheta$ denotes the boolean result of applying the valuation ϑ to c, and we say that "$c\vartheta$ holds" if $c\vartheta = \text{true}$. Note that this is a specification of S, not an algorithm for computing it!)

$$\begin{aligned} S_0 &= \{\vartheta \mid \forall c \in H_0 \; c\vartheta \text{ holds}\} \\ S &= \{\vartheta \mid \vartheta \in S_0 \wedge \forall \sigma \in S_0 \; \neg better(\sigma, \vartheta, H)\} \end{aligned}$$

We now define the locally-predicate-better and weighted-sum-better comparators. As also noted above, we use an error function $e(c\vartheta)$ that returns a non-negative real number indicating how nearly constraint c is satisfied for a valuation ϑ. This function must have the property that $e(c\vartheta) = 0$ if and only if $c\vartheta$ holds. For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate* comparator. For a domain that is a metric space, we can use its metric in computing the error instead of the trivial error function. Such a comparator is a *metric* comparator.

The first of the comparators, *locally-better*, considers each constraint in H individually to find Pareto-optimal solutions.

Definition. A valuation ϑ is *locally-better* than another valuation σ if, for each of the constraints through some level $k-1$, the error after applying ϑ is equal to that after applying σ, and at level k the error is strictly less for at least one constraint and less than or equal for all the rest.

$$\begin{aligned}
&\textit{locally-better}(\vartheta, \sigma, H) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \forall p \in H_i \ \ e(p\vartheta) = e(p\sigma) \\
&\qquad \wedge \ \ \exists q \in H_k \ \ e(q\vartheta) < e(q\sigma) \\
&\qquad \wedge \ \ \forall r \in H_k \ \ e(r\vartheta) \leq e(r\sigma)
\end{aligned}$$

Locally-predicate-better is then *locally-better* using the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not.

Next, we define a schema *globally-better* for global comparators. The schema is parameterized by a function g that combines the errors of all the constraints H_i at a given level.

Definition. A valuation ϑ is *globally-better* than another valuation σ if, for each level through some level $k-1$, the combined errors of the constraints after applying ϑ is equal to that after applying σ, and at level k it is strictly less.

$$\begin{aligned}
&\textit{globally-better}(\vartheta, \sigma, H, g) \equiv \\
&\quad \exists k > 0 \ \text{ such that} \\
&\qquad \forall i \in 1 \ldots k-1 \ \ g(\vartheta, H_i) = g(\sigma, H_i) \\
&\qquad \wedge \ \ g(\vartheta, H_k) < g(\sigma, H_k)
\end{aligned}$$

Using *globally-better*, we now define *weighted-sum-better* by selecting a particular combining function g. The weight for constraint p is denoted by w_p. Each weight is a positive real number.

$$\begin{aligned}
\textit{weighted-sum-better}(\vartheta, \sigma, H) &\equiv \textit{globally-better}(\vartheta, \sigma, H, g) \\
\text{where } g(\tau, H_i) &\equiv \sum_{p \in H_i} w_p e(p\tau)
\end{aligned}$$

Different constraint solvers find solutions for different comparators. In our work to date, the two solvers we use that accommodate soft constraints are DeltaBlue, which finds locally-predicate-better solutions, and Cassowary, which finds weighted-sum-better solutions. (When there are infinitely many solutions, as in the above example with the two medium constraints $x = 0$ and $y = 0$, Cassowary has the additional property that it finds solutions that "tilt" toward one constraint or the other, so that it would find either the solution $x = 0$, $y = 10$ or else $x = 10$, $y = 0$ — but not for example $x = 3.6$ and $y = 6.4$, even though that is also a weighted-sum-better solution.)