Concrete Type Inference: Delivering Object-Oriented Applications

A dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. (December 1995)

Ole Agesen

SMLI TR-96-52 January 1996

Sun Microsystems Laboratories

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
ole.agesen@eng.sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.
For distribution issues, contact Amy Tashbook Hall, Assistant Editor <amy.hall@eng.sun.com>.

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David M. Ungar (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell (Co-Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

L. Peter Deutsch

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the University Committee on Graduate Studies:

Abstract

Types can range from concrete, describing implementations of objects, to abstract, describing interfaces or other properties of objects. Object-oriented languages exploit this distinction to allow the same code to be reused with objects of varying forms. This *polymorphism* permits the use of objects with different concrete types, as long as they have the abstract type required by the context. For example, list-based and array-based stacks have different concrete types, but can be interchanged since they both implement the abstract type stack. Polymorphism adds expressive power, but sacrifices manifest concrete types in programs (even in statically-typed languages). In turn, application delivery suffers: lacking concrete type information, it is impossible to compile as efficiently, to check as thoroughly, and to eliminate dead code as effectively.

We have designed, implemented, and evaluated an algorithm, the *cartesian product algorithm*, that can infer concrete types of object-oriented programs. It applies the cartesian product to break the analysis of each polymorphic send into a case analysis; each case represents a monomorphic combination of actual arguments and can be analyzed precisely. The cartesian product algorithm improves precision and efficiency over previous algorithms and deals directly with inheritance, rather than relying on a preprocessor to expand it away. Our implementation handles most of the dynamically-typed Self language, including dynamically-dispatched messages, dynamic and multiple inheritance, lexically-scoped blocks, and non-local returns, although it supports the reflective features of Self only partially.

Using the inferred concrete types, a *compiler* can statically bind and inline message sends, a *browser* can follow control-flow through message sends, an *extractor* can identify the essential objects for an application and discard the rest, and a *checker* can statically guarantee the absence of message-not-understood errors. Collaborating with other researchers, we have implemented proof-of-concept versions of these type-based application delivery tools.

Measurements indicate that type inference performs well. For example, on a 167 MHz UltraSPARC™ our type inferencer, written in Self, can analyze the DeltaBlue constraint solver in 10 seconds (DeltaBlue consists of 500 lines of source code and uses an additional 900 lines from libraries). Using the inferred types, our extractor, also written in Self, takes 4 seconds to produce a shrink-wrapped DeltaBlue application. The shrink-wrapped application comprises 350 Kb, a 96% size reduction over the original 10 Mb image. Moreover, using the inferred types, Hölzle's Self compiler can compile statically and still produce native code of a quality comparable to the state-of-the-art code that it generates in its normal dynamic compilation mode. Thus, concrete type inference enables delivery of object-oriented applications.

Acknowledgments

Five years ago I moved 10,000 kilometers to join Stanford University and the Self project. It has been worth it, every year and every kilometer. In the Self system, created by all the past and present members of the Self group, I found an excellent research vehicle. In the Self group and its surrounding community at Stanford University and Sun Microsystems Laboratories, I found people who were always willing to discuss my work and offer criticism, suggestions, and encouragement. Only then, with the direct and indirect help of all these people, did this Ph.D. dissertation become possible.

David Ungar, my advisor, contributed much inspiration and set, through his own work, high standards for what constitutes good research. His ability to establish an environment in which research can happen is unsurpassed by anyone I know. So is his ability to locate good pizza.

L. Peter Deutsch, John C. Mitchell, and Mendel Rosenblum bravely volunteered to be on my reading committee at a time when little of this dissertation had been written. I owe them thanks for invaluable and insightful feedback and discussions on previous drafts of this dissertation.

I am indebted to all of the present and former members of the Self group, but in particular those whose time in the group overlapped with mine: Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, and Mario Wolczko. Working with them has been a lot of fun. Bay deserves special thanks for never complaining about sharing an office with too many pairs of running shoes, and for proof-reading the final draft of this dissertation.

Without the people at Daimi, Aarhus University’s Department of Computer Science, this dissertation might never have happened. From the outset, Ole Lehrmann Madsen and Eric Meineche Schmidt supported my idea of pursuing a degree from an American university. Later, during one of my visits to Daimi, meetings with Jens Palsberg and Michael I. Schwartzbach got me interested in type inference.

In addition to the above-mentioned persons, during the research leading to this dissertation, I have been fortunate to discuss parts of my work with Jecel Mattos de Assumpção Jr., Charles Consel, Bjorn Freeman-Benson, T.K. Lakshman, Ivan Moore, John B. Plevyak, Markku Sakkinen, Jan Vitek, and many others.

Last, but not least, Harumi deserves my thanks for all her support and understanding. I can never fully pay her back, but I’ll keep trying.

Finally, I wish to thank all the organizations that generously provided financial support to make this work possible, especially The Natural Sciences Faculty of Aarhus University, The Danish Research Academy, and Sun Microsystems Laboratories.

Table of Contents

List of Tables

List of Figures

1 Introduction

Object-oriented languages, by their very nature, pose new problems for the *delivery of applications*, the construction of time-efficient, type-safe, and compact binaries suitable for distribution to users. By allowing the programmer to write code that is more abstract and reusable, object-oriented languages forfeit the explicit specification of implementation details. Without these details, it is impossible to compile as efficiently, to check as thoroughly, or to eliminate dead code as effectively.

For example, consider the Smalltalk and C versions of a method to compute the maximum of two numbers in Figure 1. Unlike its pre-object-oriented C counterpart, the Smalltalk method says nothing about whether it operates over integers, floats, or even lists; indeed, it will work on any two objects that provide a less-than operation. When delivering an application that uses the `max` method, the C compiler can use the type declarations to type-check and optimize the `max` method; it can even determine that the less-than operation for lists is not needed by the `max` method. The Smalltalk system, lacking the type information, cannot deliver as effectively.

```
max: arg = (
  arg < self ifTrue: [
    ^ self.
  ] ifFalse: [
    ^ arg.
  ]
)
```

Smalltalk

```
int max(int arg1, int arg2) {
  if (arg2 < arg1) {
    return arg1;
  } else {
    return arg2;
  }
}
```

C

Figure 1. Smalltalk and C methods for computing maxima

Some researchers have attempted to remedy this problem by adding type declarations to Smalltalk. Unfortunately, with this approach, any gains in deliverability come at the expense of reduced reusability of code. Continuing with the example, if the arguments of `max:` were declared with a *concrete type* such as 32-bit integer, then the delivery process would be as easy as for C. However, the method could no longer be reused as broadly, e.g., floats would be excluded. On the other hand, if an *abstract type* such as `Number` were chosen, most reuse would be preserved since the method would still apply to all kinds of numbers. However, delivery would be no better off since the possibility that many different kinds of numbers may occur in `max:` must be accounted for. At best, the introduction of type declarations can only give the programmer the opportunity to choose between deliverability and reusability; it cannot provide both at once. Consequently, the delivery problem exists for statically-typed object-oriented languages like Beta, C++, and Eiffel, as well as for dynamically-typed ones such as Smalltalk and Self.

Ideally, programmers should be able to write and deliver abstract and reusable code, focusing on domain-level problems and ignoring implementation details such as whether numbers are 32-bit integers or have some other concrete type, and whether sets are represented as binary trees or have some other concrete type. To reach this goal, the concrete type information necessary for delivery of time-efficient, safe, and compact applications must be inferred automatically.

In this dissertation, we investigate algorithms to automatically infer the concrete types of an object-oriented program, and the use of such types to build programming environment tools that support application delivery; see Figure 2. For the programmer using a dynamically-typed object-oriented language, the tools provide him with the ability to deliver efficient, type-safe, and compact applications. For the programmer using a statically-typed language, the tools relieve type declarations of their role in supporting delivery, allowing the programmer to write code that is more abstract and reusable without impairing delivery.

We have designed, implemented, and tested a prototype of such a type-inference-based delivery system for the dynamically-typed object-oriented language Self. The contributions of our work include:

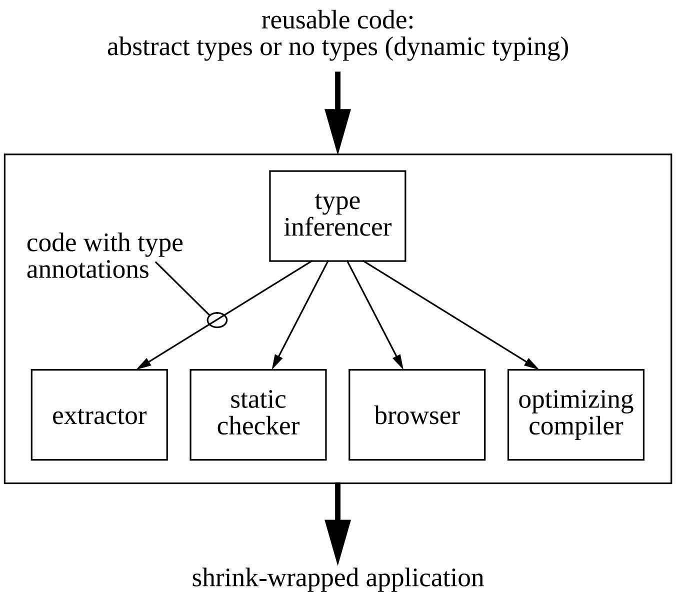

reusable code:
abstract types or no types (dynamic typing)
type inferencer
code with type annotations
extractor
static checker
browser
optimizing compiler
shrink-wrapped application

Figure 2. Type-inference-based application delivery system

- *Templates*[†], a concept allowing a high-level characterization of how a type-inference algorithm analyzes parametric polymorphism (the ability of methods to be invoked on different kinds of actual arguments, exemplified by the Smalltalk `max:` method above). We apply templates to analyze and directly compare five existing algorithms for inference of concrete types, exposing both their strengths and weaknesses.
- *The cartesian product algorithm*, a new type-inference algorithm for analyzing code with parametric polymorphism. This algorithm, motivated by the analysis of the previous algorithms, simultaneously improves precision and efficiency.
- *Type inference techniques* for analyzing important features of object-oriented languages: dynamic dispatch, inheritance, lexically-scoped blocks (closures), and non-local returns. Moreover, we have developed and implemented flow-sensitive inference for variable accesses, grouping to allow inference on objects rather than source code, and incremental inference to speed up reinferring types after program changes.
- *An extraction algorithm* that, using the types computed by our inferencer, can deliver compact shrink-wrapped applications from the Self environment. The extractor needs no programmer intervention, takes only minutes to extract, reduces the image size by an order of magnitude for medium-sized applications, and completely preserves the behavior of the application across extraction. No other delivery tool for dynamically-typed integrated object-oriented environments offers all these advantages simultaneously.
- *Other tools*. Collaborating with Urs Hölzle, we investigated type inference's role in supporting optimizing compilation of Self programs. We found type inference capable of delivering the same execution efficiency as does type feedback, a dynamic technique known to out-perform commercial Smalltalk systems significantly. In cooperation with Ole Lehrmann Madsen, we implemented a type-based static checker and browser. The checker detects where message-not-understood errors may occur. The browser can trace control flow through dynamically-dispatched sends, thereby helping the programmer navigate from the place where an error was detected back to its cause.

[†] To prevent confusion: our notion of templates differ from C++'s.

Although we have performed our research in the context of the Self language, our results have broader applicability, extending also to statically-typed languages, as argued above. In addition, we hope that our work may provide a better basis for discussing type systems and contribute to an improved understanding between the statically-typed and dynamically-typed object-oriented communities. While having different stances with regard to type declarations, both communities share the need for concrete type information and both have come to reject declarations for supplying concrete type information.

The remainder of this dissertation consists of the following parts: background (Chapter 2), type inference (Chapters 3, 4, and 5), applications of type information (Chapters 6 and 7), and conclusions (Chapter 8).

In more detail, Chapter 2 discusses general properties of types and type systems, introduces the Self language and environment, and reviews related work. Chapter 3 defines the concept of types and then focuses on type-inference algorithms for parametric polymorphism. Using templates, it reviews five existing inference algorithms and presents the cartesian product algorithm. Chapter 4 continues the discussion of type inference, addressing a broader range of issues than parametric polymorphism. Chapter 5 discusses recursive customization, an interaction between recursion and type inference that may potentially cause non-termination of type inference. Recursive customization can also affect optimizing compilation. Chapter 6 describes the application extractor in detail, and Chapter 7 outlines the other applications of type inference, i.e., compilation, static checking, and browsing. Finally, Chapter 8 concludes by taking a broader look at what has been accomplished and laying out possible directions for future work.

The glossary on page 167 briefly defines the most important technical terms used in this thesis.

2 Background and related work

This chapter presents background material. First, Section 2.1 discusses type systems and polymorphism. Subsequently, Section 2.2 gives a brief introduction to the Self language and programming environment, emphasizing the aspects that are particularly relevant to the later chapters. Finally, Section 2.3 reviews related work.

2.1 Type systems and polymorphism

Before considering type inference per se, a broader look at type systems is in order. We characterize type systems along two dimensions: concrete to abstract (Section 2.1.1) and general to specific (Section 2.1.2). Then, using these dimensions, we characterize polymorphism in object-oriented languages (Section 2.1.3) and discuss open- versus closed-world assumptions in relation to types (Section 2.1.4).

2.1.1 Concrete and abstract types

Types can range from concrete to abstract, as illustrated in Figure 3.

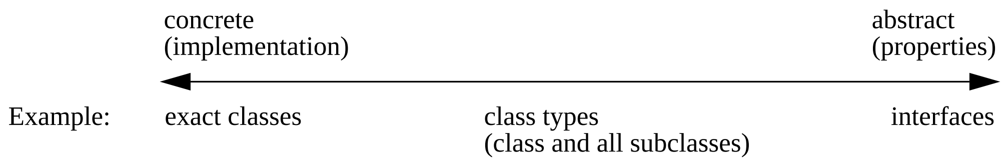


Figure 3. A spectrum of type systems: from concrete to abstract

Concrete types describe implementations of objects, including machine-level properties, such as how objects are laid out in memory and the code they execute to perform operations. In many object-oriented languages, including Smalltalk [46], C++ [110], Beta [77], Eiffel [79], Simula [38], Trellis [104], and CLOS [14], but not Self [117], classes[†] define implementations of objects. Hence, the concrete type of an object is the exact class that it has been instantiated from (we emphasize "exact" to avoid confusion with the "class and all subclasses" notion of types used in Simula, C++, Eiffel, Beta, and other statically-typed languages; see below). Concrete types provide detailed information, e.g., they distinguish even between different implementations of the same behavior.

For expressions that may evaluate to instances of more than one class and for variables that may hold instances of more than one class, concrete types are not single classes but *sets of classes*. Consider the classes in Figure 4 and these examples:

- An expression that always returns instances of class `ListStack` has the concrete type {`ListStack`}. The type {`ListStack`, `Queue`} (reading "," as "or") is also legal for the expression, but the latter type provides less information since it leaves open the possibility that the expression might return `Queue` instances in addition to the `ListStack` instances. The expression does not, however, have the concrete type {`Stack`}: this type describes an expression that returns instances of the exact class `Stack`[‡].
- A variable that holds instances of either class `ListStack` or class `ArrayStack` has the concrete type {`ListStack`, `ArrayStack`}.

[†] Some languages use a different terminology, e.g., Beta uses the term *pattern* instead of class.

[‡] To avoid confusion, readers accustomed to the style of type declarations found in C++ should note that the concrete type {`Stack`} has a different meaning than the C++ type declaration `Stack`. The former type denotes instances of the exact class `Stack` only, whereas the latter type includes instances of subclasses of `Stack`.

Section 3.1.2 revisits the definition of concrete type in the context of the classless Self language. Until then, it suffices to think of a concrete type as a set of exact classes.

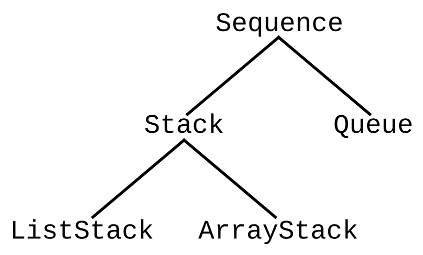


Figure 4. Class hierarchy

Abstract types describe properties or invariants of objects. For example, Emerald's [13] *interface types* list a set of operations that objects must support and the types of arguments and results of the operations. Abstract types characterize the externally observable behavior of objects and need not distinguish between different implementations of the same behavior. Returning to our example, both `ListStack` and `ArrayStack` instances may have an interface type like

`Stack_IF = [push:elmType→void; pop:void→elmType]`†.

`Stack_IF` specifies that objects have a `push` and a `pop` operation, but leaves unspecified how these operations are implemented. Any object that defines a `push` and a `pop` operation, regardless of how these are implemented, has the abstract type `Stack_IF`.

Smalltalk and Self, while having no linguistic notion of type, nevertheless employ interface types: dynamic type checking in these languages entails verifying that objects "understand" (i.e., define a method for) the messages that they receive. In fact, interfaces are so important that they are introduced on the very first page of Goldberg and Robson's Smalltalk description [46].

Many statically-typed object-oriented languages, including Simula, Beta, C++, and Eiffel use *class types*‡. The property that a class type expresses is: "being an instance of the given class or one of its subclasses." Thus, referring again to the classes in Figure 4, the class type `Stack` includes instances of the classes `Stack`, `ListStack`, and `ArrayStack`, whereas the class type `ArrayStack` includes `ArrayStack` instances only. In fact, for a given program (under the closed-world view discussed in Section 2.1.4), a class type can always be expanded to a concrete type, e.g.:

class type `Stack` = {`Stack`, `ListStack`, `ArrayStack`}

class type `ArrayStack` = {`ArrayStack`}.

We place class types between the concrete types and abstract types in Figure 3. First, class types are not fully abstract: they partially reveal the implementation of objects. For example, if class `Stack` defines an instance variable `size`, it can be asserted that any object of class type `Stack` has this instance variable††. Beta's "inner" mechanism for method combination [77] likewise implies that methods defined in class `Stack` extend to any object of class type `Stack`. Second, class types are not fully concrete; knowing a class type for an object only implies that the specified

† This example is not in Emerald syntax.

‡ Other statically-typed languages, including Strongtalk [19] and TOOPL [20], use variations of interface types.

†† However, a few languages permit subclasses to override instance variables with get/set methods.

class is a superclass of the object's class. The object may have more instance variables and operations than the class type reveals.

Since class types include the specified class and *all* its subclasses, their resolution is coarser than that of concrete types which specify sets of exact classes (i.e., includes no subclasses except those explicitly included in the set). Consider again Figure 4. The concrete type {`ArrayStack`, `Queue`} has no equivalent class type. The closest larger approximation is the class type `Sequence`; however, specifying that an expression evaluates to an object of class type `Sequence` is less precise than specifying that it evaluates to an object of concrete type {`ArrayStack`, `Queue`} since the class type does not rule out that the expression returns a `ListStack` instance.

Our type-inference system, aiming to support delivery of applications, infers concrete types. With the detailed information that concrete types provide, more powerful delivery tools can be constructed than would be possible based on, say, programmer-specified class types.

2.1.2 General and specific types

Types characterize how code may be used (and reused). A given piece of code can often be annotated with several types. Consider the simplest possible example, the identity function, in Self defined by:

```
    id: x = ( x ).
```

The *most-general type* for the identity function is `any`→`any`[†]. This type describes the *legal use* of the identity function: no matter which object `id:` is invoked upon, it succeeds, returning an object of the same kind. (Here, `any` denotes a type that any object conforms to. If the type system employs class types, `any` is the root class; if the type system employs interface types, `any` is the empty set of operations).

In the context of a specific program that invokes `id:` only on floating point objects, `id:` could also be annotated with the type `float`→`float`. This type describes the *actual use* of the identity function and is the *most-specific type* possible with respect to the given program. In a different program that invokes `id:` on both floats and integers, `id:` has the most-specific type {`float`→`float`, `integer`→`integer`} (reading "," as "or"). Thus, most-specific types are relative to a particular program; they imply a *closed-world* view (see section Section 2.1.4). Figure 5 illustrates how type annotations may range from most-specific to most-general.

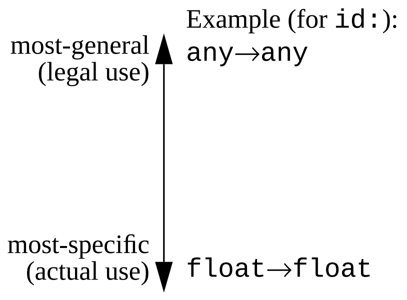


Figure 5. The specific/general dimension of type systems

Most-general and most-specific types serve different purposes. For example, to determine how a piece of code may be used, the programmer can study most-general types; to determine how a particular program uses the code, most-specific types may provide a better starting point. To support delivery of applications, types should be specific (and concrete, as argued above) to allow the delivery tools to work with stronger invariants. To illustrate, consider a method that swaps the two top-most elements on a stack:

[†] For simplicity, we ignore the type of the receiver since it plays no role in this example.

```
swap: aStack = (
  |e1. e2|
  e1: aStack pop.
  e2: aStack pop.
  aStack push: e1.
  aStack push: e2.
  self.
).
```

This method can be applied to both `ArrayStack` and `ListStack` instances. A compiler translating `swap:` can generate more efficient code under the assertion that the argument has type {`ArrayStack`} than under the assertion that it has type {`ArrayStack`, `ListStack`}. Using the former more specific type, the compiler can inline the `ArrayStack` versions of `pop` and `push:` straightforwardly; using the latter more general type, the compiler must emit machine code that selects between the `ArrayStack` and `ListStack` implementations of `pop` and `push:` at execution time.

Our type-inference system, described in the following chapters, was designed to infer specific types to augment its usefulness for delivery.

Although the end-points in the specific/general dimension provide the most information, computability and efficiency concerns of type inference may necessitate conservatism by pushing the inferred types slightly towards the center of the axis.

- If the goal is to obtain most-specific types, the inferencer may have to settle with slightly more general types to be practical. Consequently, if the types drive compilation, occasionally optimization opportunities may be missed because of conservatism of the inferred types.

- If the goal is to obtain most-general types, the inferencer may have to settle with slightly more specific types to be practical. Consequently, if the types determine how code may be used, occasionally reuse of code may be prevented because of conservatism of the inferred types.

We stated above that our type inferencer infers most-specific types. More precisely, it infers as specific types as possible under certain conservative assumptions about control flow (e.g., the type inferencer may fail to discern the direction of conditional branches).

While all four combinations of specific/general and abstract/concrete are meaningful, in practice (specific, concrete) and (general, abstract) types seem more useful than the other two combinations. Henceforth, we will often write “concrete type” to denote a concrete and specific type, and “abstract type” to denote an abstract and general type, reserving the longer phrases for the few cases when they are required for disambiguity.

2.1.3 Polymorphism

Object-oriented languages derive significant expressive power from *polymorphism*, the ability to use an object of any concrete type, as long as it satisfies the abstract type required by the context. For example, `ListStack` and `ArrayStack` instances can be used interchangeably in contexts expecting stacks, regardless of whether this code was written with one or the other implementation of stacks in mind. The expressiveness promotes reuse of code and allows programmers to think in terms of abstract domain-level types rather than concrete implementation-level types. However, as will become clear in the following chapters, polymorphism also represents a particularly hard challenge for type inference.

For later purposes, we need to distinguish between two forms of polymorphism:

- *Parametric polymorphism* refers to the ability of methods to be invoked on objects of different concrete types. For example, the `swap:` method in Section 2.1.2 exhibits parametric polymorphism since it applies to both `ListStack` and `ArrayStack` instances.

- *Data polymorphism* refers to the ability of a slot (or variable) to hold objects of multiple concrete types. For example, if `link` objects are used to build lists of integers and lists of floats, the `contents` slot in `link` objects exhibits data polymorphism.

To analyze polymorphic code precisely, both parametric and data polymorphism must be explicitly addressed by the analysis algorithm. Moreover, although the two kinds of polymorphism have common aspects, current state-of-the-art type inference applies different techniques to address the two kinds of polymorphism.

2.1.4 World views

Under a *closed-world* assumption, all the classes making up an application are available for inspection and analysis in advance of executing the application. In contrast, an *open-world* assumption permits extensible applications for which the full set of classes are statically unknown: the application may be extended with new classes in the future. The open-world assumption is the more powerful model for application development, but also the harder for program analysis.

Since concrete types explicitly enumerate exact classes, they effectively impose a closed-world assumption. For example, if a formal argument of a method has been annotated with a concrete type such as `{ListStack}`, the type annotation prevents the method from being applied to instances of new extension classes. In contrast, abstract types can co-exist with an open-world assumption, since they do not imply an explicit enumeration of classes. For example, annotating the formal argument of a method with a class type such as `Stack` permits the method to be applied not only to instances of the currently known subclasses of class `Stack`, but also to instances of subclasses of `Stack` with which the application may later be extended.

As argued in Section 2.1.2, by accepting a closed-world assumption, code can be annotated with more specific types than under an open-world assumption. In our work on type inference for supporting delivery, we adopted the stronger closed-world assumption. Then, types can be concrete and specific, maximizing their effectiveness for supporting delivery. However, it should be remembered that this gain in deliverability comes at the cost of accepting the closed-world view. Should one want to extend an application, the extension must be absorbed into the closed world and the application re-analyzed and repackaged for delivery with the extension included. In the future, this trade-off may need to be revisited.

The current trend in the industry is towards applications that are open in the sense that they can be controlled from other applications and can call out to “plug-in” modules whose implementation, and perhaps even interface, may be only partially known statically. This increasing openness, to a significant degree motivated by the object-oriented perspective that computer systems consist of cooperating entities (objects), is to some degree at odds with the results and approaches developed in this dissertation under the closed-world view. Resolving or limiting this potential conflict is future work, but we can envision two approaches: first, we can seek to relax the closed-world assumption under which this dissertation’s work was done; second, we can develop ways to manage the openness so that we may successfully apply closed-world techniques to parts of open systems.

2.2 Self

Self is more than just a language. It is the combination of a language, a set of reusable objects, a programming environment, a user interface, and a virtual machine. In the following two subsections we review some of the more unique features of the Self system, focusing on the aspects that are particularly relevant, unusual, or challenging for a type-inference system. Section 2.2.1 considers the core language itself, whereas Section 2.2.2 examines the system as a whole. Since Self in many regards resembles Smalltalk, we describe it with reference to Smalltalk.

2.2.1 The Self language

Self [108, 117] was conceived and designed as a simpler yet more expressive alternative to Smalltalk [46]. Self is based on prototypes [17] and has no classes or meta-classes. Simplicity is ensured through uniformity: all data are objects, all expressions are message sends, and computation is performed solely by sending messages.

Like Smalltalk programs, Self programs contain no static type declarations. Instead, type-correctness is verified during execution, i.e., Self is *dynamically typed.* Dynamic typing maximizes expressiveness by eliminating the need for more restrictive static type checking. It is an important factor in attaining the highest possible degree of polymorphism and, therefore, code reuse. Since Self code contains no type declarations there are no a priori restrictions on how it may be reused—as long as the context respects the (implicit) invariants in the code.

Message sends are always *dynamically dispatched:* the method invoked depends on the dynamically computed receiver of the message. Dynamically-dispatched sends give the programmer expressive power, but are more difficult to analyze than statically-bound procedure calls. To infer the type of a dynamically-dispatched send, *two* problems must be solved. First, the dispatch must be resolved, i.e., the type inferencer must compute the set of methods that the send may invoke. Second, each of these methods must be analyzed to determine the kinds of objects they may return. In comparison, when analyzing a statically-bound procedure call, the former problem is trivial.

In Self, unlike Smalltalk, variables are always accessed by sending dynamically-dispatched messages, regardless of whether the variable is in the receiver or some other object. The uniform use of message sends blends state and behavior: from the sender's perspective, it is impossible to tell whether a message is implemented by reading a variable or performing a computation. For this reason, the word *slot* is used to denote both variables and methods. When an object receives a message, the matching slot is located. If the slot contains a method, it is executed and its result becomes the result of the send. If the slot contains a reference to a non-method object, this object becomes the result of the message. Self's blending of state and behavior adds expressive power, but also means that dynamically-dispatched sends are used even more frequently than in other object-oriented languages. Consequently, type inference of Self programs depends on precise resolution of dynamic dispatch to an unusually high degree.

In this dissertation, the term "send" denotes a syntactic entity, an expression. In texts referring to other languages, the term "call site" has been used with a similar meaning. However, procedure calls connote static binding, so we usually prefer the word "send" over "call." Sometimes we will say "send expression" to disambiguate the static expression from the dynamic action of sending a message. Associated with the word "call" are terms like "caller" (the procedure containing the call) and "callee" (the invoked procedure). We will usually say "the sending method" and "the invoked method," respectively.

Self has *multiple inheritance:* one or more slots in an object may be marked as parents. When an object receives a message, a lookup is performed to find the *target* slot, i.e., the slot invoked by the message. In the simplest case, the receiver object contains a slot whose selector matches the message, but if it contains no such slot, the parents of the receiver, and, if necessary, their parents, etc., are searched. If no matching slot can be located, the send fails with a message-not-understood error. If a matching slot is found, no matter where, it is evaluated as described above. In Self, any object may be used as a parent. Neither the language definition nor the implementation places any restrictions on the structure of the inheritance graph: repeated inheritance (inheriting from the same object several times) and cyclic inheritance graphs are both allowed. For a precise description of the semantics of lookup, see [4]; for historical comments about their evolution and rationale for the current semantics, see [108].

In Smalltalk, and indeed most object-oriented languages, the inheritance graph cannot change during program execution. Self does not impose this restriction. *Dynamic inheritance*, implemented via assignable parent slots, allows the inheritance graph to change during execution. Dynamic inheritance adds significant expressive power: a single assignment to a parent slot may change the future behavior of one or several objects dramatically. A further aspect of inheritance in Self is the ability to *inherit state*. This ability follows directly from the prototype model in which any object can be a parent, including those that contain state.

Many type-inference systems expand away inheritance before inference. The expansion simplifies type inference (one less language construct to handle) and improves precision (inherited methods can be analyzed in the context of a single class). However, Self's powerful inheritance semantics, and dynamic inheritance in particular, defy expansion. Consequently, we have crafted the Self type inferencer to work on unexpanded programs, tackling inheritance directly and ensuring precision through other means than expansion. In particular, to analyze dynamic inheritance, the type inferencer computes an upper bound for how much the inheritance graph may change during execution and uses this upper bound to approximate lookups through dynamic parent slots.

Following Smalltalk, Self has only a small set of basic control structures built in: dynamically-dispatched message sends, non-local returns, a `_Restart` primitive, and a few more. Other control structures such as conditional statements and while-loops are constructed out of the basic ones. Unlike Smalltalk implementations, which often hardwire the definitions of the most common control structures (including conditional statements and certain loops) to gain efficiency, the Self programmer has the liberty to modify them all at will. The consequence for type inference is that less can be taken on faith. If the Self type inferencer encounters a statement such as

```
condition ifTrue: [doSomething] False: [doSomethingElse]
```

it can *not* assume that this statement has the semantics of a conditional statement. For example, `condition` might evaluate to a non-boolean object that implements `ifTrue:False:` in a way that causes neither, either, or both block(s) to be executed any number of times, and in any order. Even if `condition` always returns a boolean, the programmer might have modified the definition of `ifTrue:False:` in the boolean objects. In contrast, if an analyzer for C encounters this statement

```
if (condition) {doSomething} else {doSomethingElse}
```

the analyzer can straightforwardly exploit many properties of `if-else` to sharpen the analysis. For example, exactly one of `doSomething` or `doSomethingElse` will be executed. For a Self analyzer to reason as precisely, it would need to prove such properties as part of the analysis.

In summary, the Self language maximizes expressive power through dynamic typing, uniform use of dynamic dispatch, powerful inheritance semantics, and programmer-defined control structures. These design choices, while a blessing for the programmer, combine to form a difficult challenge for static analysis.

2.2.2 The Self system

The Self system has evolved over several years and contains a large number of reusable objects. These objects implement general data structures such as lists, sets, points, rectangles, and unlimited-range integers (bigInts). Other objects support construction of graphical user interfaces; yet others implement an integrated, interactive programming environment, including text editors, object browsers (outliners), and debuggers [4, 119]. Together, this set of objects, comprising on the order of 10 Mb, is known as the “Self World.”

The current Self system, and the one we refer to throughout this dissertation, is release 4.0. Since the first release of the Self system, programming has shifted away from being solely a text- and file-based activity. Today programmers interact with the Self system in a number of ways. They may

- manipulate objects directly through a graphical user interface. Typical operations include cloning objects, copy/cut/pasting parts of objects (i.e., slots), adding new slots to objects, and editing method definitions.
- type expressions at a prompt and have them evaluated. These expressions are unrestricted and can perform arbitrary computation, including global changes, such as scanning the heap for all objects matching a certain criteria and adding or removing slots from them.
- invoke meta-programming tools, i.e., programs that write programs. Specific examples are Mango (a parser generator [1]) and PrimMaker (a program that can add new primitives to the Self system).
- enter data into the image—since everything is an object in Self, the distinction between data and programs is not manifest, and neither is the distinction between data entry and programming.

At any point in time, the state of a Self system is the result of its *computational history* and its primary representation is the set of objects in the image. To facilitate versioning and exchange of objects between different Self systems, the Self world contains a *transporter.* The transporter can derive *modules*, textual representations of related objects and slots, from the binary objects and a set of programmer-supplied declarations [118]. Modules can be converted back into binary objects by a parser in the Self virtual machine, but are secondary: they are generated from the binary objects and are never edited directly under normal circumstances. Instead, editing takes place directly on objects as described above, and modules are regenerated as necessary.

In summary, to fully support the exploratory programming approach, *objects are the primary representation of programs in the Self system*. Consequently, type inference must be performed directly on objects rather than on source code.

2.2.3 Self versus Dylan

The Dylan language [11] illustrates the tension between the desire to maximize expressiveness and the need for concrete type information to address pragmatic concerns such as execution efficiency and application size when delivering applications. The design goal for Dylan was to provide the programmer with a flexible dynamic language (hence, "Dylan") for "rapid creation, delivery, and subsequent modifications of (…) software." Yet, to satisfy the additional goals of being "practical on small machines" and "achieve commercial performance in production code," the designers of Dylan introduced concepts that allow the programmer to significantly limit the polymorphism of code, *sealed classes* being the most important example. A sealed class cannot be subclassed. Consequently, the class type of a sealed class is a concrete type consisting of a single exact class: the sealed class itself. Moreover, any method operating on a sealed class is monomorphic. Concrete type information was gained, but at cost in flexibility: sealed classes cannot be subclassed, and code operating on their instances is monomorphic.

Thus, the designers of Dylan accepted giving up some flexibility to improve deliverability. In contrast, the designers of Self set out to maximize expressiveness, even if this would mean giving up the ability to deliver applications as effectively. The type-inference system described in this dissertation shows that, despite the different design goal, the Self system can *still* deliver applications.

2.3 Related work

Given the broad applicability of type information in programming environments, it is no surprise that several different approaches to computing type information have been, and are being, explored. The type-inference system described in this dissertation draws upon some of these previous and contemporary approaches. However, it also relates to work that does not have the particular goal of type inference, including data-flow analysis, abstract interpretation, and some of the analyses employed in partial evaluators. We cannot possibly cover all these areas in detail, so the following review only includes projects that are particularly relevant when trying to understand the Self type-inference system in a larger context. Further discussions of related work when relevant to specific properties of the Self type inferencer can be found in the chapters that describe these properties.

We characterize the related work with respect to a number of issues, such as the kind of programming language that a type-inference method applies to and the kind of types it infers. The issues are presented in Section 2.3.1, and specific projects are reviewed in Section 2.3.2. For each project, the focus will be on a selected set of issues, typically those which were emphasized in the project, or those which reveal strong similarities or differences with the Self system. Finally, Section 2.3.3 will summarize the related work.

2.3.1 Issues

The following list of issues will be used to characterize the related work. The issues cannot always be considered independently. For example, the purpose of inferring types will influence which kind of types should be inferred. In addition to describing each issue, we indicate where the Self type inferencer stands.

The purpose of inferring types. A type-inference system cannot be evaluated without a context. The problem or problems that it is intended to solve must be taken into consideration because different problems require different levels of precision, need different kinds of types, and can tolerate different computational costs of type inference. For example, if type inference mainly supports application extraction, "overnight" performance may be acceptable (assuming that applications are only extracted occasionally). On the other hand, if type inference mainly supports optimizing compilation, faster inference is essential since compilation is a frequent task.

The overall purpose of applying type inference to the Self system, as explained in the introduction, is delivery of applications, including performing specific tasks such as static checking, browsing/debugging, extraction, and optimizing compilation.

Kind of types. Whether one wants to understand a type-inference system, apply it to solve a specific problem, or compare two type-inference systems, one of the first steps should be to identify the kind of types inferred by the system(s). Are they concrete or abstract? Specific or general? Is there more than one kind of type? For example, the Self type inferencer computes types that are concrete and specific.

Language and environment. This dissertation focuses on type inference for object-oriented languages, but since these languages share features with other languages, we include some related work done for non-object-oriented languages. Specific features of the Self language that have had significant consequences for our type-inference system are: polymorphism, dynamic dispatch, inheritance, and blocks[†]. In addition to being directly influenced by language features, a type-inference system must respect the rules of the programming environment in which it is integrated. For example:

- The Self environment emphasizes direct interaction with objects. This emphasis makes type inference a different task than if the environment had used source code as the primary representation of programs.
- The existence of a large body of code written prior to the introduction of type inference, as in the Self environment, may make it harder to introduce type inference. The inferencer must deal not only with new code written with the knowledge of the inferencer, but also must cope with old code, possibly written in styles less amenable to type inference.

Inference method. We give an operational explanation of how type inference proceeds for each system reviewed. We also consider whether type inference is *modular* (e.g., done for a method at a time) or non-modular (i.e., performed for a whole program at a time). A non-modular type inferencer operates under a closed-world assumption: the entire program must be available at analysis time. Finally we consider whether the type system relies partially on programmer-inserted type declarations or is based entirely on inference.

The Self type inferencer, as will be explained in detail in the following chapters, performs a combined control- and data-flow analysis to compute types, is non-modular, and does not need declarations (but could take advantage of them, if they were present).

Precision and efficiency. Any type-inference algorithm must approximate to remain computable, but different degrees of approximation are possible. Often, precision trades against efficiency: if one is willing to spend more CPU cycles, more precise types can be expected.

Implementation status. The present type-inference system for Self started as a theoretical project. The approach seemed promising in theory, yet when it was implemented the precision obtained was extremely disappointing. For example, while the original system could infer that `3+4` may return a smallInt, it was unable to rule out dozens of other kinds of objects, including `nil`, `true`, `false`, `byteVector`s, and several blocks. Only after significant new developments did precision improve to the present, and far better, level. Given this experience, we consider an empirical validation invaluable when dealing with anything as complex as a type system that is supposed to handle large programs, not to mention `3+4`!

The expression 3+4 is not as simple as it may seem at first glance. The Self system, like most Smalltalk systems, define several kinds of numbers: floats, smallInts, and bigInts. *SmallInts* are integers that fit in a single machine word. Their range, in the current implementation, is limited to $-2^{29} \ldots 2^{29}+1$ (30 bits) since the Self virtual machine reserves two tag bits of every 32-bit word. *BigInts*, on the other hand, need not fit in a single machine word. Their range is limited only by the available memory. Unlike smallInts, whose representation is defined by the virtual machine, bigInts are implemented entirely in Self code using smallInts, and byte vectors. When arithmetic operations on smallInts overflow, the error is caught, the smallInts are coerced to bigInts, and the overflowed operation is retried. Since the type inferencer performs no range analysis or constant folding, it cannot rule out the possibility of arithmetic overflows, even when analyzing an expression as simple as `3+4`. Accordingly, type inference of `3+4` involves the bigInt implementation and therefore is far from trivial.

[†] While most object-oriented languages include polymorphism, dynamic dispatch, and inheritance, blocks are presently less universal, being absent from both C++ and Eiffel. However, blocks are not Self or Smalltalk specific: patterns in Beta and lambda expressions (closures) in functional languages both resemble blocks closely.

2.3.2 Projects

The following projects are ordered primarily according to the approach they have taken. For example, the data-flow inspired projects of Barnard, Vitek, and Pande are presented in sequence. The secondary ordering criterion has been chronological.

2.3.2.1 Hindley and Milner

The best known type-inference algorithm is probably the one developed by Hindley for combinatory logic and easily reformulated for the lambda calculus [55]. This algorithm was later rediscovered and extended in a programming language context by Milner [80]. Cardelli's report [21] gives a practical introduction. The algorithm is today an integrated part of any ML compiler. We review the basic system in which there are three kinds of types[†]:

- *Basic types*, e.g., `Int` and `Bool`, describe the primitive objects in the language.
- *Type variables*, usually written with a Greek letter, α, β, γ, ..., are used in polymorphic types. They can be thought of as universally quantified, ranging over types.
- *Function types*, written $\rho \rightarrow \sigma$, where ρ and σ are types. They describe the type of a function that maps objects of type ρ to objects of type σ. For example, `Int`→`Int` is the type of the factorial function and $\alpha \rightarrow \alpha$ is the type of the identity function (it maps an argument of any type α to a result of the same type).

Type inference in the Hindley/Milner system is done for one function at a time, proceeding bottom-up on expression trees. Initially, a type is assigned to all leaves in the expression tree. If a leaf is a constant, it is assigned the corresponding basic type. If a leaf is a variable access, the type assigned is the type variable for the accessed variable. In the bottom-up phase, a set of inference rules is used to deduce the types of composite expressions from their components and to restrict the types of the components to ensure that they fit together in the composite expression. For example, if the expression is the function application `f(x)`, and we currently know that `f` has type α, and `x` has type β (perhaps it is the first use of `f` and `x` we have encountered, so they are still unrestricted), the inference rule for function applications allow us to deduce that the type of `f` must be a function type from the type β to some other type δ, i.e., $\alpha = \beta \rightarrow \delta$. Later, an expression `x+1` may be encountered. Since the primitive (built-in) function `+` has type `Int`×`Int`→`Int` or `Float`×`Float`→`Float`, the former case must apply (the type of `1` is `Int`, ruling out the latter case), and we can conclude that the type of `x` is `Int`, i.e., β = `Int`. The type inferencer combines this information with what is currently known about the type of `f`, which therefore becomes: α = `Int`→δ.

Type inference fails if conflicting information is encountered in this process. For example, if one part of an expression leads to the conclusion that the type of `x` is `Int`, and another part demands that `x` has type `Bool`, type inference fails—the expression is ill-typed. An ill-typed expression does not necessarily give a run-time type error if it is executed, but it *may.* On the other hand, Milner proved that if type inference succeeds, the expression will execute without type errors.

An elegant way to implement the combination and restriction of type information employs a unification algorithm on type expressions. For practical ML programs, this algorithm executes efficiently, but Kannelakis, Mairson, and Mitchell have shown that the worst-case time complexity for typing ML is exponential in the nesting depth of `let` declarations [66].

A comparison between the type-inference system implemented for Self and the Hindley/Milner system reveals significant differences in several areas. We consider three areas of particular importance.

- *Kind of types inferred.* Hindley/Milner-style type inference computes *principal types* or most-general types[‡]: types specify only enough to rule out run-time type errors. This property holds because types (for non-constants) start out as completely unrestricted type variables and are restricted only as necessary to eliminate the possibility of type errors. Intuitively, the bottom-up phase of type inference allows the maximal amount of polymorphism to

[†] One may think of *type schemes* as a fourth kind; they describe ML's `let`-bound identifiers. Type variables in the type of a `let`-bound identifier (unlike a lambda-bound one) can be instantiated differently at each occurrence of the identifier, allowing application of a polymorphic function to different types of arguments in the same context [21]. We ignore let-polymorphism here.

bubble up to the surface, i.e., the interface of the function. For example, take the function that doubles its argument by adding it to itself:

```
double(x) = x+x.
```

In a hypothetical language with several kinds of numbers (floats and smallInts, say) that all implement arithmetic, Hindley/Milner-style type inference would infer the type `number`→`number`, since this type is the least restricted one that describes `double`[†]. This generality is good news for the programmers, since only the necessary restrictions are imposed on the type of the argument, leaving them with as much polymorphism and potential code reuse as possible. But the generality is bad news for application delivery: even if a *particular program* only invokes `double` on float arguments, this information cannot be deduced from the inferred type. Consider compilation. With the principal type, the compiler must compile `double` under the assumption that the argument can be either an integer or a float, yielding less efficient code than if it could specialize the generated code to the float case. The Self type inferencer takes the opposite approach. Types start out maximally restricted as empty sets of classes, and are *relaxed* to accommodate uses as they are encountered. For `double`, the initial type of the argument `x` is {}, but if a send is encountered that invokes `double` on a float, the type is relaxed to {`Float`}. If all uses of `double` supply float arguments, the final type inferred for `double` will be {`Float`}, giving the compiler a better opportunity to optimize.

- *Inference method.* Hindley/Milner-style type inference analyzes a function at a time, i.e., is modular. Consequently, separate compilation of functions is supported. In contrast, the Self type inferencer is non-modular. It performs a global flow analysis to infer types and therefore does not support separate compilation. Intuitively, the global analysis is necessary because without analyzing the entire program, it cannot be ruled out that, say, `double` is invoked on integers.

- *Typing of variable accesses.* The Hindley/Milner system is flow insensitive: all accesses to a given variable share the same type. The Self inferencer, on the other hand, is flow sensitive: two different expressions that both read the same variable may have different types. The extra flexibility of the latter approach allows significantly better precision when analyzing Self code. We can think of two reasons why this improvement would be considerable in Self, but may not be in ML. First, ML emphasizes *binding* of variables whereas Self puts a higher emphasis on assignment (an intervening assignment is the typical reason that two reads of the same variable return different kinds of objects). Second, unlike ML, the Self system evolved for a long period without consideration for static typing. Inferring types for an existing body of code written under liberal circumstances requires more flexibility than if the type system is in place from the beginning to prevent, say, programmers from writing code that stores first a boolean and then a file object in the same variable when they could just as well use two different variables.

Milner's original type-inference system covered the functional core of ML. It has since been extended to deal with additional language features. Tofte added support for state and assignment [116]. Cardelli, Rémy, Wand, and others studied how to add records modelling objects both with and without subtyping (expressed as record extension, i.e., the addition of fields) [22, 98, 122]. Hense and Smolka studied inference of principal types for O'small, a minimal object-oriented language with inheritance [54]. Curtis proposed constrained quantification to increase the expressive power for typing programs combining ML-style polymorphism and object-oriented subtyping [36]. Cartwright and Fagan proposed *soft typing* [24]: instead of rejecting programs that cannot be statically typed, their type inferencer inserts type-checking operations at the untypable points, making the modified program statically typable, but at the cost of no longer eliminating the possibility of type errors at run-time (the type-checking operations may fail). Aiken, Wimmers, and Lakshman also studied soft typing [10].

[‡] The generality of a type can be formalized using the concept of *substitution*: the replacement of free occurrences of a type variable by a type. A principal type for an expression is a type with the property that all other types for the expression can be obtained from the principal type via substitution. Damas and Milner proved that the inference algorithm computes principal types [39].

[†] This example is hypothetical because ML requires that overloaded operators can be statically resolved by the type inferencer. Thus, + must be resolved at type-inference time to either $+_{\texttt{smallInt}\times\texttt{smallInt}\rightarrow\texttt{smallInt}}$ or $+_{\texttt{float}\times\texttt{float}\rightarrow\texttt{float}}$.

Since we are mainly interested in concrete and specific types, these extensions are outside the scope of the present review. The interested reader may find a survey by Tiuryn covering some of the developments up to 1990 in [115]. Mitchell gives another overview in [81].

2.3.2.2 Borning and Ingalls

Borning and Ingalls carried out one of the first attempts at adding types to Smalltalk [18]. They designed and implemented a combined type declaration and inference system for Smalltalk-80. They primarily intended to use the types to inform the programmer about potential message-not-understood errors at "compile time" rather than execution time. They listed two additional applications of their type system as future work. One was to use types to produce faster and more compact code by allowing better factoring of information. The other was to use type information "…to produce application modules containing just the code needed to run a particular application," i.e., to extract applications from a Smalltalk image.

In Borning and Ingalls' system, a type is a set of messages, each of which specifies the types of the arguments and the result. An object *has* a type if it understands all the messages in the set. However, types by default, and in the majority of cases, correspond to classes. For example, they write the type of an instance of class `Point` as `<Point>`. Since their paper gives few details of when types and classes differ, in this review of their type system, we will assume that classes and types map one-to-one.

The correspondence between types and classes organizes types in a hierarchy parallel to the class hierarchy: one type is a supertype of another type if its class is a superclass of the other's class. For example, consider this conditional expression taken from Borning and Ingalls' paper:

```
x = y ifTrue: [3] ifFalse: [3.1416].
```

It has type `<Number>` because the type of `3` is `<Integer>`, the type of `3.1416` is `<Float>`, and the nearest common superclass of `Integer` and `Float` is `Number`. Because union types such as `<Integer> or <Float>` are not included in their system, some precision is lost every time two expressions with different types merge (`3` and `3.1416` above). In the above example, the loss is minor, and most programmers would probably find the type `<Number>` to be quite natural for a conditional statement that can return either integer or floating point numbers. However, consider this conditional expression instead:

```
x = y ifTrue: [3] ifFalse: ['horse'].
```

It will be assigned the type `<Object>` because class `Object` is the nearest common superclass of `Integer` and `String`. The difference between knowing that an expression returns "an object" versus an integer or a string is significant. After all, *any* non-failing expression in Smalltalk returns an object, so inferring the type `<Object>` conveys very little information.

In Borning and Ingalls' system, type checking is modular, done on a method at a time. This decomposition is enabled as follows:

- All formal arguments and results of methods are declared with a static type. This requirement shields callers from the internals of callees.
- Variables are declared with a static type. This requirement shields variable reads from variable writes. (Variables local to a method need not have a type declaration, but in this case Smalltalk's scoping rules ensure that all reads and writes are found within the same method, so modularity is not affected).
- Overriding methods must return subtypes of the overridden methods. This requirement shields callers from overriding.

With these three additions to Smalltalk-80, type checking becomes a local problem. To infer the type of a message send, and check that it does not result in message-not-understood, it suffices to recursively check the receiver and arguments, then look at the type declaration of the method that the send invokes and verify that the types of the actual arguments match the declaration of the formal arguments.

Unfortunately, the resulting type checker is unsound, a fact which Ingalls later became aware of and points out in [61]. The type checker may accept a program that causes a type error when executed. While the paper by Borning and Ingalls is not completely clear, it seems that their system uses a *covariant* rule for types of variables and method arguments. For example, the type `<self class>`, which denotes the type of the receiver, results in covariance. Covariance of method arguments causes a method `foo:` in a subclass to be applicable to *fewer* actual arguments than the `foo:` method in the superclass (the formal argument's type in the subclass is more restrictive). Consequently, if an instance of the subclass is used where an instance of the superclass was expected, a type error may result. Several concrete examples of how this problem arises can be found in a later paper by Cook on Eiffel's type system [33]. At the time when Borning and Ingalls wrote their paper, the properties of covariance and contravariance were not well understood in the object-oriented community. Today, however, following the publication of several papers [22, 23, 25, 33, 76] that analyze this aspect of object-oriented languages, these complications are better understood.

In the common case, when Borning and Ingalls' types correspond to classes, they are semi-abstract, analogous to the class types found in Beta, C++, or Eiffel (see Section 2.1.1). The significant reliance on declarations means that the programmer can push the type system in the direction of specific types (by "overconstraining" in the type declarations) or general types (by carefully choosing the least restrictive type declarations). However, both extremes come at a cost: specific types restrict polymorphism whereas general types are less useful for optimization and extraction.

Borning and Ingalls implemented and tested their type declaration and inference system on a small part of the Smalltalk system, but did not use it during a large project. Unfortunately, they give no results on the degree to which their system could type-check existing Smalltalk code. However, based on the work later done by Graver and Johnson (see Section 2.3.2.4), we suspect that non-trivial modifications would be required in many areas of the Smalltalk system before the code could be accepted by Borning and Ingalls' type inferencer and checker.

In retrospect, the major limitation in Borning and Ingalls' system is their notion of type, and the dilution of type information that happens whenever two differently-typed expressions meet and are assigned the type of their nearest common superclass. In their system, the precision loss can be kept somewhat in check by the "fire-walls" that declarations build. However, trying to extend their system to use more inference and fewer declarations will likely be difficult, because imprecisions can migrate further and accumulate to an unacceptable level.

2.3.2.3 Suzuki

Suzuki had observed that late binding of message sends, or dynamic dispatch, was a source of inefficiency in the then-current Smalltalk-76 system. He also noted that late binding could impede program understanding by making it hard for the programmer to find out which method a message invokes. In response to these problems, Suzuki designed a type-inference system that he envisioned could be used to eliminate dynamic dispatches and annotate programs for improved readability [111].

In Suzuki's system, the basic types are *sets of classes*. Thus, the number of types is *exponential* in the number of classes in the program, whereas in Borning and Ingalls' system the number of types is *linear* in the number of classes (or possibly polynomial when counting their parameterized types). When more types are available, more precise invariants can be expressed using them. For example, using Suzuki's types, the Smalltalk conditional expression

```
x = y ifTrue: [3] ifFalse: ['horse']
```

can be assigned the precise type {`Integer`, `String`}, whereas Borning and Ingalls' inferencer had to assign it the type `<Object>` and therefore lost valuable information.

In addition to the basic types, Suzuki's type system has function types to describe methods. Function types resemble those found in Milner's system: they describe the argument and result types of a method. When a method takes more than one argument, the type on the left hand side of the arrow is a product type (each component in the product is the type of an argument).

Suzuki's system is a pure inference system without any type declarations. His inference method is based on the Hindley/Milner approach. Since Smalltalk-76 has features not found in functional languages, Suzuki had to extend the Hindley/Milner type-inference algorithm in several regards. He added support for assignable variables (both temporary and global) and relaxed the typing rules for conditional statements to permit different types for each of the branches. His most important extension dealt with dynamically-dispatched sends. Here, Suzuki faced the difficulty of two mutually dependent problems:

- To infer the type of a send, it is necessary to conservatively approximate the set of methods it may invoke.
- To approximate the set of methods that a send may invoke, the type of the receiver expression, which is often also a send, must be known.

We will refer to the former problem as *dynamic dispatch resolution* or simply dispatch resolution. Thus, the difficulty that Suzuki faced can be rephrased as a mutual dependency between dispatch resolution and type inference: to infer types, dynamic dispatches must be resolved; but to resolve dynamic dispatches, types are necessary.

Suzuki's system broke this mutual dependency by initially inferring types under the maximally conservative assumption that a send can invoke *any* method with a matching selector, no matter where in the system the method is defined. Of course, this pessimistic assumption leads to less precise type inference. To gain precision, Suzuki proposed *iterating* the analysis, using the types inferred in the previous iteration to resolve dynamic dispatch less pessimistically in the current iteration.

Interestingly, Suzuki took the opposite approach to resolve dynamic dispatch than we have taken in the Self type-inference system (see Section 4.1). The Self inferencer assumes that a send invokes *nothing* until this is proven wrong whereas Suzuki's inferencer assumes that a send may invoke *anything* (with a matching selector) until this is proven wrong. It turns out that Suzuki's approach is less powerful intuitively because the very assumption that a method may be invoked can be reason enough that it cannot be ruled out! To illustrate, assume that the program being analyzed contains a global variable `v` and a send `v foo`. Assume furthermore that class `A` defines a `foo` method as follows:

```
foo = ( v: self. “do the foo thing” ).
```

To further sharpen the example, assume that `v` initially contains `nil`, that there are no assignments to `v` other than the one in the `foo` method, and no sends with the selector `foo` other than the send `v foo`. Thus, there is actually no way for the program to invoke the `foo` method since the only accessible send of `foo` will always have `nil` as the receiver. When analyzing the send `v foo` the first time, Suzuki's inferencer assumes that all `foo` methods may be invoked. Consequently, the type inferred for the variable `v` is {`A`, `UndefinedObject`[†]} since `foo` in class `A` assigns an `A` instance to `v`, and `nil` is the initial contents of `v`. In the second iteration, Suzuki's inference algorithm must again assume that the `foo` method can be invoked because the first iteration's inferred type for the receiver expression `v` includes class `A`. Additional iterations do not improve the precision. Hence, the iteration has *converged*, but with a suboptimal result. In summary, Suzuki's resolution of dynamic dispatch sometimes cannot recover fully from the initial weak assumption.

The specific problem of dynamic dispatch resolution is discussed in detail in Section 4.1. Section 4.1.3 discusses the general problem of mutually dependent analyses and characterizes two general solution techniques: iteration and integration. Suzuki applied the former technique to resolve dynamic dispatch, whereas the Self type inferencer uses the latter technique. Throughout Chapter 4, several more problems amenable to either technique are discussed.

Suzuki also applied iteration to infer types for global and instance variables. Like his iterative dispatch resolution, his iterative variable analysis proceeds from most pessimistic to less pessimistic. In the first iteration, each variable *V* has a type containing all classes in the system. Thus, any read (use) of *V* also obtains this type. In subsequent iterations, *V* has a smaller type, determined as the union of the types inferred in the previous iteration of all assignments to *V*. Like the iterative dispatch resolution, this iterative analysis technique for assignable variables sometimes cannot fully recover from the very pessimistic assumptions of the first iteration. In comparison, as will become clear, the Self type inferencer does not iterate to analyze variables and avoids the initial pessimism by inferring types of variables from empty sets towards progressively larger sets.

Suzuki implemented only a simplified version of his inference system in which instance- and global variables always have the type consisting of all classes in the system. The implementation was attempted tested on the classes that

[†] In Smalltalk, `nil`'s class is `UndefinedObject`.

implement numbers and arithmetic in Smalltalk-76, but Suzuki gives no specific results, except that he had problems fitting his system within the limited heap of Smalltalk-76 (Smalltalk-76 limited the number of live objects to 65536 [71]). Later, Suzuki abandoned the inference approach and worked with Terada on a declaration-based type system for Smalltalk-80 [112]. Thus, the effectiveness of this type-inference approach for Smalltalk-76 remains undocumented in practice. For later Smalltalk systems or the Self system, which contain many more classes and methods, we believe that Suzuki's approach may be insufficiently powerful. In particular, the iteration from most pessimistic to less pessimistic for both dispatch resolution and typing of instance- and global variables becomes progressively more strained as the analyzed systems grow. To give specific numbers, Smalltalk-76 contains only 60 classes whereas version 4.0 of the Self system contains some 1,800.

2.3.2.4 Graver and Johnson

Headed by Ralph Johnson, the Typed Smalltalk project aimed to improve Smalltalk by adding a static type system [48, 49, 62, 63]. Primarily, they wanted the types to enable compiler optimizations and statically detect message-not-understood errors, but Graver [48] lists secondary uses of the types as well, including the provision of documentation to the reader of programs and as an aid in the programming process. To some extent, their goals had built-in conflicts. For example, to optimize programs, one would want types to be concrete and specific, whereas to use types as a specification and documentation, one would want them to be abstract and perhaps general to avoid both needless detail and hampering of code reuse. Graver seems to acknowledge this discrepancy when he writes [48] (p. 3):

> … we introduce a type-inference algorithm that can effectively infer over 90% of the type-declarations required by the compiler. Unfortunately, there is a trade-off between optimization and type-inference. The type-inference algorithm computes the most general type for a method or variable even though the actual use of the method or variable may be much more specific. In order to maximize optimization, the more specific type should be supplied by the user.

There are several kinds of types in Typed Smalltalk; see Table 1. To some degree, the different kinds of types are associated with different concepts in Smalltalk, but there are cases where a given entity can have more than one kind of type. For example, the argument of a method `star:` could be annotated with:

- a class type such as `Integer` to express that `star:` should only be called with integer arguments,
- a union type such as {`Integer`, `Float`}, to express that both integers and floats are acceptable,
- a signature type such as `{Self+Self->Self, Self power: Integer -> Self}` to express that `star:` can be called with any object that understands + and `power:` with the given argument and result types.

The type `Self`, used above, denotes the type of the receiver in a method. In Typed Smalltalk, inheritance is expanded away before inference starts. The expander replaces the type `Self` with the class type of the class it occurs in. Thus, the type inferencer never encounters the type `Self`.

Kind of type	**Description**
class type	consists of a class name and a list of type parameters (parameterized classes were mainly used for collections)
union type	is a set of types
block type	specifies the types of the block's arguments and its return type
message type	consists of a message selector, a receiver type, a list of argument types, and a result type
method type	consists of a message type and some constraints
signature type	is a set of message types: the set of messages that a class type must understand to be in the signature type

Table 1. Overview of the types in Typed Smalltalk

Typed Smalltalk employs a combination of type declarations and type inference to determine type information. Instance variables, class variables, and global variables must have type declarations. Local variables, formal arguments, and methods may or may not, and expressions rarely have. Whenever types are undeclared, they are inferred. With the required type declarations in place, the remaining types can be inferred one method at a time[†]. Four steps are required to infer the type of a method (and in the process, a type for every expression in the method body):

- Step 1. The method's parse tree is constructed.
- Step 2. The parse tree is changed to a *type tree* by replacing all leaf nodes with their respective types. For example, if the parse tree contains the integer `5`, it is replaced by its class type `Integer` in the type tree. References to formal arguments and temporary variables with type declarations are replaced by the declared types. When a formal argument or temporary variable has no type declaration, a *type variable* is allocated for it and used in place of the type declaration. The type variables, like those in Hindley/Milner's inference system, are initially unconstrained, i.e., they may represent any type whatsoever.
- Step 3. The type tree is *reduced* by performing an abstract interpretation of the method body over the domain of types. During the reduction process, a type is inferred for every expression, and type variables are constrained according to the use of the entities they represent.

  For example, to type a message send, the receiver and arguments are first reduced to obtain types for them. Then the receiver type is used to resolve the dynamic dispatch. For each method that may be invoked, it is verified that the actual argument types of the send are *subtypes* of the formal argument types of the method. The subtype rules are relatively straightforward. For example, a union with fewer members is a subtype of a union with more members, and a class type is a subtype of any union type in which it is included. The subtype relation degenerates to type equality in the case of array elements because these can be both read and written (i.e., have both covariant and contravariant accessor methods). If the required subtype relationship holds for all methods and arguments, the message send can be typed and its type is the union type of the return types of the invoked messages.

  Sometimes dynamic dispatch cannot be resolved because *concrete* type information is missing for the receiver. For example, knowing a signature type for the receiver does not allow dispatch resolution with high accuracy, since any class type matching the signature type must be assumed possible. When this happens, the remainder of Step 3 is deferred and will later be completed in the context of each call site invoking the method. By then (it is hoped), context-specific information will be available to overcome the missing information. In comparison, the Self type inferencer *always* defers type inference, since it never analyzes a method unless it has found it through some call invoking it.
- Step 4. A method type is assigned to the method, based on the result of the reduction. If the reduction step did not reduce the tree to a single node, the left-over tree forms the constraints of the method type. It is these constraints that are later reduced in the context of each caller; see Step 3.

It was never a goal of Typed Smalltalk to avoid type declarations completely. Rather, the goal was to infer *most* types, but when necessary or beneficial use type declarations. In retrospect, this partial reliance on declarations combined negatively with the mixture of concrete and abstract types in the same type system. On one hand, the compiler needs concrete and specific types to support optimizations. On the other hand, to retain polymorphism and preserve the spirit of the dynamically-typed Smalltalk-80 system, the programmer should stick with abstract and general type declarations. Thus, the programmer is caught in a dilemma between supplying very specific declarations (little reuse possible) and very general declarations (few optimizations possible).

Following Milner, Graver and Johnson coupled type inference and type checking, i.e., the verification of the absence of type errors. This coupling lends strength to the type-inference system: the fact that a method has a type is sufficient to guarantee that it executes without type errors. However, the coupling also makes type inference more difficult: to infer types, one must statically prove that type errors cannot happen during execution. In our work, we have taken the opposite approach of Graver and Johnson: type inference and type checking are fully separated. It is possible to infer

[†] Although performed for one method at a time, type inference is not modular in a strict sense, since the copy-down elimination of inheritance necessitates re-checking of methods when they are inherited.

a type that indicates that a runtime error may, or even *will*, happen at a given point. We delegate type checking to a separate *checking tool* that runs independently of type inference, but uses the inferred types.

The extra degree of freedom provided by the decoupling of type inference and type checking is particularly important when adding a type system to an existing body of code. With thousands of lines of existing Smalltalk code (or Self code) written without regard for static typing, there *will* be code that cannot be shown type-correct. The problem of typing an existing body of code should be seen in contrast to the *easier* task of introducing a type system from the very beginning. In the latter case, the designers of the type system are in control. They can influence language design and programming style to maximize the performance of the type system and type inferencer. The danger, of course, is that they may design a type system that pleases only the type checker, not the programmers. Bracha and Griswold also observe that the existing body of code is a great challenge when retrofitting a type system to Smalltalk [19]. For their statically-typed Smalltalk dialect, Strongtalk, they refrained from type-checking the existing Smalltalk code. Instead, they suggest "downward compatibility" as a better model: Strongtalk code can be trivially converted to Smalltalk code, but not vice-versa.

Graver and Johnson implemented their type-inference system, but apparently were only moderately successful in applying it to the standard Smalltalk image [63]. They report measurements on compiling and executing small programs such as one that sums all integers between 1 and 10,000, but they give no large examples.

2.3.2.5 Barnard

Barnard's dissertation compares two different analysis systems, both of which compute type information for Smalltalk programs [12]. The first system is a cleaned-up and simplified version of Graver and Johnson's type-inference system from Typed Smalltalk (blocks and primitive failure code were omitted). The second system is a data-flow analysis system (for introductions to the field of data-flow analysis, see [9, 83]). Barnard's immediate goal was to compare the systems' relative strengths and weaknesses. Here we focus on his data-flow system.

Barnard observes that data-flow analysis of object-oriented languages cannot be separated from control-flow analysis, because data flow determines control flow at dynamically-dispatched sends. This dependency renders most of the traditional data flow texts inapplicable to Smalltalk (and for that matter to any object-oriented language) because they assume the existence of a control-flow graph *prior* to the start of data-flow analysis. To overcome this difficulty, Barnard's analysis, like the Self type inferencer, is a combined control- and data-flow analysis.

In Barnard's flow analysis, unlike traditional flow analyses and unlike the Self type-inference system, there is no main method or notion of program. Instead, any method in the Smalltalk image can play the role of a main method. This decision matches the interactive programming model of Smalltalk where nothing prevents the programmer from sending any message to any object, i.e., a program execution can, at least in principle, start anywhere. However, permitting sending any message to any object does not mean that it will happen during the execution of some particular program. Barnard is aware of the potential precision loss arising from this assumption, as he writes:

> … this will inevitably result in numerous 'bogus' control paths being followed, with the relevant 'non-bogus' paths counting as a minority of the total number of paths analysed.

He envisions addressing this problem in a user interface to the flow analysis system, but gives no details.

Barnard concludes, based on his theoretical study of the two systems, that the flow analysis system can provide the same error checking capabilities as the type-inference system while avoiding some of the drawbacks (such as rejecting "good" code). Unfortunately, Barnard never implemented the two systems, so it remains to be seen if his theoretical observations can be validated empirically.

2.3.2.6 Shivers

Olin Shivers worked on Scheme and was not specifically interested in type inference [106, 107]. Rather, he wanted to compute control-flow graphs for Scheme programs. Control-flow information, he argues, is a prerequisite for applying the standard data-flow analysis algorithms developed to optimize Fortran-like languages, without which Scheme will never execute as efficiently as Fortran. Once control-flow information is available, type information can be computed using data-flow techniques.

Precise control-flow graphs for Scheme programs are difficult to compute because higher-order functions may contain function calls whose targets are not obvious. For example, the `map` function, which invokes a function on all the elements of a list, may be defined by:

```
(define (map f L)
  (if (null? L) nil (cons (f (car L)) (map f (cdr L)))))
```

Determining the control-flow through `map` is difficult because the function (closure) `f` invoked by the underlined call is not manifest: `f` is a parameter of `map`. This problem is equivalent to determining which method a dynamically-dispatched send may invoke:

- To analyze Scheme programs, closures must be tracked to the call sites invoking them.
- To analyze object-oriented programs, objects must be tracked to the places that send messages to them.

The analogy between a functional language with higher-order functions and an object-oriented language with dynamic dispatch suggests that object-oriented programs could be analyzed by transforming them into Scheme programs, applying Shivers' algorithm, and mapping the results back. While true in principle, programming style differences may make functional and object-oriented programs quite different in practice. For example, in Self, *every* message send is dynamically dispatched, whereas in typical Scheme programs, most function calls are "statically bound" (i.e., they do not invoke a function which has been passed as an argument). Indeed, in the `map` function above, the targets of all the calls but the underlined one can be trivially determined. In contrast, in the equivalent Self program all of `if`, `null?`, `nil`, `cons`, `car`, `map`, and `cdr` would be dynamically-dispatched sends.

Shivers emphasizes the formal development of his analysis. He starts with a denotational semantics for continuation passing style (CPS) Scheme and derives from it a non-standard semantics that captures the relevant control-flow property. The non-standard semantics does not represent a computable algorithm, so the next step is to abstract it, i.e., introduce conservative approximations to make it computable. Shivers describes several possible abstractions, representing different trade-offs between analysis cost and precision. His simplest algorithm, called "0th-order control flow analysis" (0CFA), corresponds approximately to Palsberg and Schwartzbach's algorithm (see Section 2.3.2.9) with the program expansions switched off. The characteristic property of 0CFA is that each method is analyzed exactly once; it is a monovariant analysis[†]. Shivers likewise defines 1CFA, a polyvariant analysis in which each method is analyzed once per syntactic call site, as in Palsberg and Schwartzbach's analysis.

Functional and object-oriented languages differ considerably in the emphasis that they put on state and side-effects. Shivers models assignments in the weakest possible way: all addresses are merged together into a single abstract address. This means that once something is stored anywhere in memory by an assignment, every place which reads from memory will see all stored values. He argues that in well-written Scheme programs, the role of assignment and side-effects should be minor. Suffice it to say, this argument does not hold true for most object-oriented programs.

Based on his control-flow analysis, Shivers presents several data-flow analyses for Scheme, including constant propagation and a type-inference system. The type-inference system collects type information from three sources: primitives, conditional statements, and type declarations. The type information is propagated through the flow graph using a technique he calls reflow analysis. At this point, however, support for state and assignments has been dropped.

Shivers implemented a prototype of his system, but never integrated it with a compiler. The prototype was not programmed in a scalable way, and could only analyze small programs such as the iterative factorial function (not including any bigInt implementation). While the analyzer produced good results on the small test programs, it is still an open question how well the approach scales to larger polymorphic programs.

2.3.2.7 Vitek, Horspool, and Uhl

Vitek, Horspool, and Uhl present a theoretical data-flow analysis system for a small object-oriented language [120]. In many regards, it is fair to think of their language as the core of Smalltalk, but in other regards it is quite different. Most notably, their language has built-in control structures (`while-do` and `if-then-else`) and no blocks. A

[†] In Section 3.2.2, we distinguish more precisely between monovariant and polyvariant analyses.

program consists of a finite number of classes and a main method outside of any class. They suggest two primary uses for the information that their analyzer computes: elimination of dynamic dispatch when their analyzer can tell the exact class of a receiver, and compile-time garbage collection by extending their analysis to compute life-time information for objects.

Their analysis is specified as a set of data-flow equations over a lattice of *abstract object graphs*. An abstract object graph approximates the state of the heap of a running program at a point in time. It consists of *abstract objects* and links between them. An abstract object is the analysis-time representation of one or more concrete (run-time) objects. The links represent object references. The data-flow system's meet operator merges the set of links for the two abstract object graphs to be joined. After initializing the flow value at all program points to the minimal object graph $\perp$, the data-flow equations can be solved by an iterative process to yield an abstract object graph for each program point.

Computing an abstract object graph for each program point is more costly than employing a single shared object graph for all program points (as does the Self type inferencer). This is particularly true since each object graph represents the entire heap of the program, i.e., it may be large, and there can be thousands of program points. The advantage of computing multiple object graphs is improved precision because variable accesses can be typed flow-sensitively. For example, if some access to a variable always retrieves floats whereas another access to the same variable always retrieves integers, there is a chance that the analysis can discover it (rather than conservatively reporting that both accesses may retrieve both kinds of numbers). In Section 4.5, we discuss flow-sensitive analysis in the context of the Self type-inference system.

The mapping of concrete objects to abstract objects may be varied to achieve different cost/precision trade-offs when analyzing code with *data polymorphism* (see Section 2.1.3). The simplest analysis has one abstract object per class in the program. Accordingly, the analyzer cannot distinguish between different instances of the class. Vitek, Horspool, and Uhl outline a more precise analysis in which instances of the same class are mapped to the same abstract object only if they are also allocated at the same program point (i.e., allocated by the same `new` statement). The motivation is that one allocation point may create objects that are used in one particular way (say, lists of integers), whereas another allocation point may create objects of the same class, but use these in a different way (say, lists of booleans). By having different abstract objects for each allocation point, it is hoped that the analyzer can avoid merging the different uses (in which case it would see lists of mixed integers and booleans, say). While this idea can sometimes improve precision, it can also be defeated easily. For example, in the Self system there is (essentially) only *one* allocation point, the method `clone` which invokes the primitive `_Clone`[†]:

```
    clone = ( _Clone. "Clone the receiver." ).
```

Since there is only one allocation point, the more precise analysis degenerates to the simple analysis with one abstract object per class. To overcome this problem, one could trace backwards in the call graph and discriminate abstract objects not by the `_Clone` point, but by the `clone` point, i.e., the point that invokes the above method. But even this refinement may fall short. For example, `set` objects in Self store their elements in vectors. When a `set` is cloned, it recursively clones the element vector. Thus, all element vectors of sets are created at the same point within the `set` clone method, again defeating the allocation point discrimination. We discuss data polymorphism and the role of allocation points in more detail in Section 4.3.

To account for *parametric polymorphism* (see Section 2.1.3), Vitek, Horspool, and Uhl's analysis system is polyvariant. They apply the concept of *call strings* [105] to control when a separate analysis of a method should be split off. The call string for a program point consists of the set of methods on the call path from the main method to the program point. Two sends invoking the same method share an analysis of it if their call strings are equal. However, this analysis may be infinite since the number of call strings is unbounded in the presence of recursion. To ensure finiteness, call strings are truncated to length *p* by dropping a prefix. Now, the maximal number of call strings is finite (but still large: exponential in *p*), and two sends invoking the same method share the analysis if the last *p* methods on

[†] There are a few more allocation points, since special objects are cloned using other primitives than `_Clone`. For example, vectors are cloned using `_Clone:Filler:` which takes both a length and an initial element for the elements of the vector. However, the vast majority of objects are created by the `clone` method shown above.

the call paths leading to the two sends are the same. Call strings of length one are isomorphic to the code duplication approach of Palsberg and Schwartzbach; see below. We discuss the call string approach in detail in Sections 3.2.3 and 3.2.4.

Vitek, Horspool, and Uhl never implemented their analysis algorithm. Consequently, in the paper they show only small example programs (a dozen or so lines) analyzed by hand. Thus, the algorithm's performance on large programs remains unresolved.

2.3.2.8 Pande and Ryder

Recently, Pande and Ryder have made progress on adapting data-flow techniques to C++ [89]. They state, and we agree, that the first problem to solve is that of computing concrete type information, since without it many other analyses fall apart due to the lack of precise control-flow information for virtual calls (i.e., dynamically-dispatched sends). Type analysis for C++ is both harder and easier than for Self. Contributing to making it harder is the fact that C++ is not type and pointer safe. In particular, Pande and Ryder argue that to do a good job of type analysis in the presence of multi-level pointers (i.e., pointers to pointers), the problems of alias analysis and type inference cannot be separated. On the other hand, analysis of C++ is easier than analysis of Self in other areas. For example, C++ has no closures, and many sends, if not most, are in fact statically bound in typical C++ programs.

Pande and Ryder's work is still in the early stages, but they have an implementation that handles a large subset of C++ (the features not supported are recursive data structures, multiple inheritance, pointers to functions, union types, exception handling, and `setjump/longjump`). They report on applying the algorithm to five test programs, ranging in size from 140 lines to 970 lines, and have found that except for one pathological case, approximately 90% of all virtual calls could be statically bound, because a unique receiver type was inferred. It should be cautioned, however, that their largest benchmark contains only 39 virtual calls. (The density of virtual calls is much lower in C++ than in Self, since many calls and all variable accesses in C++ programs are statically bound.)

Our survey of related data-flow work has focused on recent projects that applied data-flow analysis to determine types in object-oriented programs. However, for non-object-oriented programs, this use of data-flow analysis goes back further than the projects reviewed above. Tenenbaum's dissertation [114] from 1974 applies data-flow analysis to determine types in SETL programs. The SETL language has built-in overloaded operators and no type declarations. For example, the "`+`" operator adds integers, concatenates tuples, and forms unions of sets. Tenenbaum's analysis propagates definitions of objects forward and requirements backward. The forward propagation extracts type information from creation of objects; the backward propagation extracts type information from use of objects. For instance, the statement `y:=x+1` propagates forward the fact that `y` is an integer (when one of the operands of "`+`" is an integer, the other must also be, and the result will be an integer), and propagates backward the requirement that `x` is an integer (if `x` is not an integer the program will fail). Tenenbaum's analysis does not address procedure calls directly, although his experimental implementation would work for procedures when their arguments were declared with types. Jones and Muchnick [65] and Kaplan and Ullmann [67] later generalized and improved the precision of this forward/backward analysis.

The extensive work done on constant propagation within the data-flow analysis framework also relates to type inference. Constant propagation and type inference both discover facts (constants and types, respectively) and propagate them forward through the data flow of the program as far as possible. A recent paper by Wegman and Zadeck [124] surveys the most important constant propagation algorithms and presents a new more efficient algorithm for non-object-oriented languages. Following Wegbreit's program analysis algorithm [123], Wegman and Zadeck's most powerful constant propagation algorithm uses the discovered constants to perform dead code elimination during analysis (e.g., one branch of a conditional statement with a constant test is dead). Dead code elimination increases the number of constants since assignments in dead code cannot tinker with variables (thus, more variables remain constant). As will become clear in Section 4.1, this use of the discovered constants to limit control-flow paths resembles the Self type inferencer's use of inferred types to limit control-flow paths through dynamically-dispatched sends. Since dynamic dispatch probably occurs more frequently in object-oriented programs than do conditional statements with constant test expressions in non-object-oriented programs, the integration of control-flow analysis and type inference for object-oriented languages is even more important than the integration of control-flow analysis and constant propagation for non-object-oriented languages. In Section 4.1.3, we discuss two approaches for combining such mutually dependent analyses.

2.3.2.9 Oxhøj, Palsberg, and Schwartzbach

The type-inference system described in this dissertation is based directly on the work done initially by Palsberg and Schwartzbach and later extended and implemented in cooperation with Oxhøj [85, 86, 88]. Working on a Smalltalk subset without blocks and meta-classes, they define types as sets of classes and give an algorithm for inferring types. Type inference, like Barnard's flow analysis, combines control- and data-flow analysis, although Oxhøj, Palsberg, and Schwartzbach present it in terms of *conditional constraints*, rather than as a data-flow analysis (the conditions and constraints capture control- and data-flow properties for the program):

- Derive a set of conditional constraints from the program.
- Solve the constraints, in [85] by a polynomial-time fixed-point computation.
- Map the solution back onto the program to get type information for all variables and expressions in it.

Oxhøj, Palsberg and Schwartzbach employ several program expansions to improve precision:

- Inheritance is expanded away to reanalyze inherited methods in the context of each class inheriting them.
- Methods are duplicated for each syntactic send with the given selector to analyze code with parametric polymorphism more precisely.
- Classes are duplicated for each allocation point to analyze code with data polymorphism more precisely.

The method duplication is equivalent to Vitek, Horspool, and Uhl's call strings with $p = 1$ and to Shivers 1CFA, and the class duplication accomplishes the same as Vitek, Horspool, and Uhl's allocation-point-specific abstract objects. In Section 3.2, we demonstrate that some of these expansions provide limited benefits and incur a high cost, a fact that Oxhøj, Palsberg, and Schwartzbach also point out: in the worst case, each expansion may square the size of the program to be analyzed [86].

Our type-inference system is based directly on Oxhøj, Palsberg, and Schwartzbach's system, but extends it in several ways. Table 2 shows the most important differences. We describe selected aspects of Oxhøj, Palsberg, and Schwartzbach's system in detail in Chapter 3.

Property	**Oxhøj, Palsberg, and Schwartzbach's system**	**Present author's system**
language	toy language: Smalltalk subset	real language: Self
environment	source code based	integrated; analysis of heap of objects through *grouping*
inheritance semantics supported	single, class-based	multiple, dynamic, inheritance of state (prototypes)
treatment of inheritance	expanded away before analysis	analyzed directly
analysis of parametric polymorphism	static expansion w.r.t. call sites: 1-long call strings	adaptive: cartesian product algorithm
analysis of data polymorphism	static expansion w.r.t. creation sites	none implemented; can be added independently
control structures in the language	built in	implemented using messages, blocks, and non-local returns
analysis of variable accesses	flow insensitive	flow sensitive
type inference and type checking	coupled	separated

Table 2. Palsberg and Schwartzbach's system versus the present author's

The papers by Oxhøj, Palsberg, and Schwartzbach describe an implementation of the algorithm and show the results of analyzing several small programs. However, since they worked on a toy language, the available programs were necessarily of limited size and written specifically for the purpose of illustrating the performance and limitations of type inference. In Chapter 3, we report on applying a variant of Oxhøj, Palsberg, and Schwartzbach's algorithm to Self programs.

2.3.2.10 Phillips and Shepard

Phillips and Shepard generalized Oxhøj, Palsberg, and Schwartzbach's algorithm by allowing for more than one level of expansion. More precisely, they added two integer parameters, p and k, to allow the user to specify the trade-off between precision and analysis time. Both parameters determine call string lengths:

- p controls the precision when analyzing code with parametric polymorphism. p is the call string length used when deciding whether two calls invoking the same method can share an analysis. This parameter is equivalent to Vitek, Horspool, and Uhl's p.
- k controls the precision when analyzing code with data polymorphism. k is the call string length used when deciding whether two allocation points can create abstract objects of the same type. This parameter generalizes the single level of allocation point discrimination used in both Vitek, Horspool, and Uhl's system and Oxhøj, Palsberg, and Schwartzbach's system.

Phillips and Shepard implemented their system in Smalltalk and were consequently able to test it on realistic code. However, the implementation was not highly optimized, making type inference slow and restricting it to small programs. They report that type inference for the expression `3+4` (including the implementation of bigInts) took approximately 30 minutes when $p = k = 0$, i.e., no expansions were performed. Increasing the precision to $p = k = 1$, i.e., to a level comparable to that of Oxhøj, Palsberg, and Schwartzbach's system, caused inference time to go up to over 10 hours. While Phillips and Shepard's did not optimize their implementation, we think it still provides a valid empirical demonstration of the cost of using non-adaptive[†] program expansions to improve precision, a topic we will return to in the next chapter.

2.3.2.11 Plevyak and Chien

At the same time that the present author worked on extending Oxhøj, Palsberg, and Schwartzbach's constraint-based analysis to the full Self language, Plevyak and Chien were independently adapting it to Concurrent Aggregates [30], a dynamically-typed, single-inheritance, Scheme-based, concurrent language. Plevyak and Chien use type inference to support optimizing compilation, and have demonstrated through a complete implementation that their algorithm can infer types with high precision [93, 94, 96].

Plevyak and Chien pioneered the use of iteration to improve the precision of constraint-based analysis. Iterative analysis works as follows. Each iteration performs a full analysis of the program. The first iteration analyzes the program with no expansions. Subsequent iterations are guided by the results of the previous iteration and introduce expansions as necessary to avoid imprecisions observed in the previous analysis. Plevyak and Chien use iteration to address both parametric and data polymorphism. We defer a more precise characterization of the iterative analysis technique to Sections 3.2.7 and 4.1.3. By then we will have established a framework that allows the iterative algorithm to be described precisely and compared directly with the cartesian product algorithm, the algorithm used in the Self analyzer.

The next project we describe also focuses on compilation of dynamically-typed object-oriented programs. However, it takes a radically different approach than the type-inference algorithms described thus far.

2.3.2.12 Hölzle

The newest Self compiler, designed and implemented by Hölzle, is based on *type feedback,* which can be thought of as dynamic type inference. Type feedback extracts type information from a program execution and feeds it back to the compiler to allow it to optimize the program further [58, 59]. To extract type information, the program must be

[†] Adaptive type inference is defined in Section 3.2.5.

compiled and executed with special monitoring code inserted. The monitoring code observes which kinds of objects occur as receivers of message sends during execution.

The types in Hölzle's system are concrete: they are sets of maps. Maps [28] are an implementation-level abstraction in the Self system. A map corresponds approximately to a clone family (see Section 3.1.2). In general, two objects share a map if they have the same set of slots with the same contents of constant slots but possibly different contents of assignable slots.

Interestingly, type feedback computes *unsound* types: since the type of an expression is determined by observing a finite number of evaluations of that expression, it cannot be ruled out that new kinds of objects may occur in the future. The compiler tolerates unsound types by inserting "uncommon branches," which are traps that invoke the compiler if a hitherto unseen kind of object should later show up. When invoked by a trap, the compiler generates code for the uncommon case, replaces the trap with a direct call to the new code so that future invocations avoid the trap in this case, and branches to the new code. Other uses of type information, such as application extraction, cannot tolerate unsound types as easily: once an application has been shipped to the end-user without, say, hash tables, it may be too late if hash tables are needed after all.

In Section 7.3, we report on a study that directly compares the usefulness of type feedback and type inference for optimizing compilation.

2.3.3 Summary of related work

We have focused primarily on type inference. Even so, the reviewed projects span a large space:

- *Purpose of type inference.* We have seen type inference applied to generate documentation and specification of code (Milner), to rule out type errors (e.g., Suzuki), and to optimize code (e.g., Plevyak and Chien).
- *Kind of types.* We have covered principal types (Milner), semi-abstract class types (Borning and Ingalls), and several projects inferring concrete types (e.g., Plevyak and Chien). We have also reviewed a project employing a mixture of abstract and concrete types (Graver and Johnson).
- *Language and environment.* The presented systems target functional languages such as ML (Milner) and Scheme (Shivers), the statically-typed object-oriented language C++ (Pande and Ryder), and several dynamically-typed object-oriented languages, including Concurrent Aggregates, Smalltalk subsets, Smalltalk-76, Smalltalk-80, and Self.
- *Inference method*. We have reviewed inference systems employing unification (Milner), abstract interpretation (e.g., Shivers), data-flow analysis (e.g., Barnard), constraint-based analysis (e.g., Oxhøj, Palsberg, and Schwartzbach), and a dynamic technique (Hölzle).
- *Amount of type declarations.* Among the described inference systems, some rely almost completely on type declarations (Borning and Ingalls), some have a partial reliance (Graver and Johnson, Pande and Ryder), and others are pure inference systems (e.g., Milner, Hölzle).
- *Implementation status.* The reviewed systems range from purely theoretical (e.g., Vitek, Horspool, and Uhl), to partially implemented (e.g., Suzuki), to systems in regular use by a large community (Milner).

Table 3 gives a more complete summary of the covered projects. For reference, the Self type inferencer is included in the table. It will, of course, be described in detail in the following chapters, as will some of the other projects. The most important lessons that we learned from the studied projects, and which we applied when designing the Self type inferencer, are:

- State explicitly whether the primary use of the type information requires abstract or concrete types. In particular, if concrete types are needed (say for optimization), don't mix abstract types into the cocktail.
- Do not ask programmers for type information to support application delivery. They will face a dilemma between giving very tight specifications to obtain large benefits from the types versus very loose specifications to retain maximal polymorphism and reusability of their code.

- If an existing body of code must be handled, separate type inference from static verification of the absence of type errors. There *will* be code which cannot be shown type-correct statically, but we want type information for it nevertheless.
- A flow-sensitive approach to typing variable accesses permits more flexibility and precision. This flexibility may be particularly necessary if an existing body of code must be typed.

Project	**Main purpose**	**Kind of types**	**Language, environment**	**Inference method**	**Implementation status**
Hindley, Milner, others	type checking, documentation	principal types	ML	bottom up, unification based	fully implemented, in regular use
Borning & Ingalls	static checking	semi-abstract; type = class	Smalltalk-80	declarations and local inference	implemented; little practical experience
Suzuki	enable optimizations	concrete; type = set of classes	Smalltalk-76	generalized Hindley/Milner	tested on number classes
Graver & Johnson	optimization, static checking	both concrete and abstract	Typed Smalltalk	declarations and abstract interpr.	implemented
Barnard	compare type inf. and data flow	concrete; object abstractions	Smalltalk	two systems: inference & flow	not implemented
Shivers	compute control flow	n/a	Scheme	abstract interpretation	prototype implem.; small examples
Vitek	eliminate dyn. dispatch, GC	concrete; abstract object graphs	toy language, Smalltalk subset	data-flow analysis	not implemented
Pande & Ryder	compute control flow	concrete; type = set of classes	C++ subset	data-flow analysis	early stages; analyzes small prog's
Palsberg, Schwartzbach, & Oxhøj	optim., dead code elim., annotation	concrete; type = set of classes	toy language, Smalltalk subset	constraint-based analysis (flow)	implemented
Phillips & Shepard	optimization	concrete; type = set of classes	ParcPlace Smalltalk	parameterized "Palsberg et al."	implemented, but not optimized
Plevyak & Chien	optimization	concrete; type = set of classes	Concurrent Aggregates	iterative flow analysis	implemented
Hölzle	fast optimizing compilation	concrete, unsound; type = set of maps	Self	monitor program execution	fully implemented, in regular use
Agesen	application delivery	concrete; type = set of groups	Self	flow analysis	implemented

Table 3. Summary of related work

3 Type inference

Constraint-based analysis, the type-inference approach applied in this dissertation, was first described for an idealized object-oriented language by Palsberg and Schwartzbach [86]. We adapted their "basic" algorithm to Self, but found it unable to infer types with a useful degree of precision for typical Self programs. For example, one of the first programs subjected to type inference was the factorial method, which in the Self system is defined for integer receivers by:

```
factorial = (
  self<=1 ifTrue: [1] False: [self * predecessor factorial].
).
```

When the basic algorithm was applied to the expression `50 factorial` it inferred this type:

{smallInt, bigInt, collector, memory, memoryState, byteVector, mutableString, immutableString, float, link, list, primitiveFailedError, time, vector, sequence, false, true, nil, [blk_1], [blk_2], …, [blk_{13}]}.

When invoked on an integer receiver, the `factorial` method returns either a smallInt or a bigInt (see page 12); there are no other possibilities. Thus, out of 31 different kinds of objects deemed possible by the basic algorithm, 29 are in fact impossible and cause considerable harm by diluting the accurate information contained in the inferred type (after all, the basic algorithm did get both smallInt and bigInt right).

Upon studying the `factorial` case in detail, we found the reason for this extreme lack of precision to be the basic algorithm's weak handling of polymorphism. Other researchers have also been aware of the shortcomings of the basic algorithm, although perhaps the Self system with its high degree of polymorphism offers a particularly striking demonstration. Indeed, Palsberg and Schwartzbach suggested an improved algorithm in the same paper that introduced the basic algorithm. Since then, several distinct improvements have been published. In this chapter, we present a total of six algorithms: the basic and five improved algorithms (see Table 5 on page 36 for an overview). The improvements, described approximately in order of increasing power, pertain to how code with parametric polymorphism is analyzed. We defer the discussion of data polymorphism, the other major challenge for type inference, to the next chapter.

Of the five improved algorithms, we devote particular attention to the last, the cartesian product algorithm. This algorithm is our original contribution. It computes cartesian products to break the analysis of polymorphic send expressions into monomorphic cases that can be analyzed both precisely and efficiently. Moreover, the cartesian product algorithm achieves conceptual simplicity because the monomorphic case analysis obviates the need for the expansions and iteration that other algorithms rely on to obtain precision.

Although many of the algorithms we present leading up to the cartesian product algorithm are the works of other researchers, our presentation of them is more than just a review: we introduce the concept of "templates", and use it to express all the algorithms in a concise and uniform manner. The coherency thus achieved facilitates direct comparisons between the algorithms to a degree not previously possible. In particular, we assess their strengths and weaknesses by applying them to concrete code fragments. Often, code fragments can be chosen that delineate the abilities of the algorithms: e.g., one algorithm cannot precisely analyze a piece of code, but the next more powerful algorithm can. We use code fragments found verbatim in the Self system or similar to code found there. Hence, any difference observed in the performance of the algorithms will likely be significant not just on test programs but, in general, when inferring types for real programs.

After studying the algorithms, we have formed two conclusions. First, designing an algorithm that is either efficient *or* precise is easy, but defining an algorithm that is both efficient *and* precise is hard. Second, to achieve efficiency and precision simultaneously, the analysis effort *must* be guided towards the areas of the program with the highest pay-off. Having observed how some of the "improved" algorithms managed to be both imprecise and inefficient, we define a general class of algorithms that have the potential to do better: the adaptive algorithms. Of the six algorithms we study, the three most powerful are adaptive.

In summary, the main contributions of this chapter are:

- *Templates:* a concept that permits a high-level and uniform description of the basic type-inference algorithm and five improvements.
- *Adaptive algorithms:* identification of a class of algorithms that use type information to optimize the cost/precision trade-off of type inference.
- *The cartesian product algorithm:* a new simple, efficient, and precise adaptive algorithm.

In the rest of this chapter, Section 3.1 gives basic definitions, including those of *program* and *type*, and establishes the framework in which constraint-based type inference operates. Then, Section 3.2 presents six type-inference algorithms, including the cartesian product algorithm, gives examples of how they work, compares their abilities, and introduces adaptive algorithms. Section 3.3 draws connections between and further compares the three most powerful inference algorithms. Section 3.4 presents empirical results for several of the type-inference algorithms, obtained by applying them to analyze programs in the Self environment. Finally, Section 3.5 summarizes the main points of this chapter.

3.1 Objects, programs, and types

To develop and explain type-inference algorithms, precise definitions of programs and types are needed. Section 3.1.1 defines programs, and Section 3.1.2 defines types. Following, a brief discussion in Section 3.1.3 rounds off the definitions.

3.1.1 Programs

Section 2.2.2 described how the Self programmer constructs applications by interacting with an image of objects, and how objects, rather than source code, are the primary representation of programs. Consequently, to fully support this exploratory model of programming, objects must also be the domain on which type inference works. Given an image (heap) of objects, we define a program in the simplest[†] possible way:

Definition. A *program* is a designated method, the *main method*, and a designated object, the *main object*, in the context of an image of objects.

The main method is usually one of the methods defined by the main object, although nothing prevents it from being inherited. The meaning of this definition will become clearer once we define program executions:

Definition. A *program execution* is the computation that results from invoking the main method with the main object as the receiver.

These flexible definitions fit well with Self's approach of encouraging programmers to experiment and explore. They may write an arbitrary expression, wrap it in a method, and call it a program (of course, if the expression is truly arbitrary, the program will likely fail when executed, but it is nevertheless a program, just a failing one). Although the above definitions require that programs take no arguments, programs can still read data from files and other external devices. Moreover, support for arguments can be added straightforwardly to the definitions and will cause no difficulties for the type inferencer, provided that the concrete types of such arguments are specified. For example, C/C++ style program arguments, which have the form

```
(char *argv[], int argc),
```

satisfy this restriction.

To be meaningful, a program must be specified in the context of an image of Self objects. Except for the fact that one of them is the main object, we make no assumptions about these objects; e.g., many of them may not take part in the

[†] An even simpler notion of a program would be to define it as an expression. However, in Self there are no "free" expressions. They can only exist in methods. For example, when the programmer types an expression, the system wraps it in a "doIt" method before evaluating it.

program execution. However, by referring to an image of objects, the definitions of program and program execution assume a closed-world view. Should this context for the program change after types have been inferred, type inference must be repeated.

Our definition of program is unusual because it focuses on the very start of executions. It refrains from explicitly delineating the set of objects in the image that participate in the execution. Instead, the definition concentrates on how to start the program. To better understand the definitions of program and program execution, we can contrast them with those that apply to a traditional language like C. Likely, the C programmer perceives programs as clearly delineated: “this file, but not that one, is part of my program.” However, dynamic libraries with interdependencies slightly erode this property. For example, the C programmer may know that a program needs a certain `collection` library, but he may not know (and need not know) exactly on which other libraries the `collection` library depends. Regarding program executions, the C programmer will find our definition familiar. In C, a program execution is also defined by how to start the program: it is the computation that result from invoking a function called `main()`, usually on a pair of arguments as mentioned above.

From now on, we will use the term *target program* to refer to a program being analyzed. Often, when there is no danger of confusing the target program with, say, the program implementing type inference, we will simply say program.

3.1.2 Types

We have defined programs in terms of objects. Concrete types, or simply types, are also defined in terms of objects, although indirectly. This section gradually constructs a working definition of type.

To understand the related work, we defined a concrete type as a set of classes in Section 2.1.1. This definition does not apply to Self, however, since there are no classes in the language. As a first attempt, consider what would happen if, instead of being defined as a set of classes, a type were defined as a set of objects. While the potential would be there for very precise type inference (as precise as observing the program execute), this set-up would unfortunately not be feasible since the number of objects, and therefore the number of types, would be unbounded. To be feasible, type inference needs a more approximate notion of type. Informally, we achieve feasibility in two approximation steps:

- *Approximation 1.* We do not distinguish between an object and the clones of it that are created during execution of the target program.

With this approximation, a type is no longer a set of objects but a set of “clone families.” There are now a finite number of types, since only a finite number of objects exist when the program is started. However, the definition is still prone to making type inference inefficient. For example, if prior to execution of the target program, the image contains 10,000 point objects (perhaps part of a picture that the program displays), these will yield 10,000 clone families, all of which are similar enough that it is a waste of effort to analyze them separately. The second approximation counters this inefficiency:

- *Approximation 2.* We do not distinguish between clone families that are generated by “sufficiently similar” objects; we *group* objects according to similarity before inferring types.

Grouping makes it possible to analyze all group members “in parallel,” thereby ensuring efficient type inference. With this second approximation, we have arrived at the final definition of type: a type is a set of groups of clone families. Equivalently, a type is a set of object types where an *object type* is the union of a group of clone families. Object types are analogous to classes in a class-based language, but more general. Just as every object in a class-based language has a class (i.e., is an instance of a class), every object in Self has an object type (i.e., is a member of some set of objects that constitute an object type).

The two approximation steps commute, so it does not matter whether one or the other step is performed first; see Figure 6. Above, we first closed objects under cloning (approximation 1), and then grouped clone families (approximation 2), but we would have arrived at the same result had we first grouped objects and then closed the groups under cloning. The composite “diagonal mapping” in Figure 6 maps an object to its object type. Its inverse maps an object type to the set of all objects that have the object type.

$$\begin{tikzcd}
\text{objects} \arrow[r] \arrow[d] & \text{clone families} \arrow[d] \\
\text{groups} \arrow[r] & \text{object types}
\end{tikzcd}$$

clone closure: $\longrightarrow$

grouping: $\downarrow$

type = set of object types.

Figure 6. From objects to types

Grouping and closure under cloning, the two abstraction steps performed to get from an intractably large number of objects to a manageable number of object types (and types), commute.

To some extent, grouping can be seen as collapsing any cloning that took place before the program execution started, whereas closure under cloning collapses the cloning that will happen during execution of the target program. For example, if the image contains 10,000 initial point objects, many of these were probably cloned from the prototypical point.

More formally, assume that we are given an image *IM* that contains a program to be analyzed. We denote the objects in the image by $\omega_1, \omega_2, \ldots, \omega_n$, i.e.,

$$IM = \{\omega_1, \omega_2, \ldots, \omega_n\}.$$

For example, it may be the case that ω_{21} is `true` and ω_7 is `nil`. One of the objects, ω_1, say, is the designated main object, and one of the methods is the main method. Because the objects $\omega_1, \omega_2, \ldots, \omega_n$ are created before the program execution starts, we call them *initial objects.* During execution of the program, additional objects are created by cloning initial objects or other objects that have been recursively cloned from initial objects. (The type-inference system can be extended to handle object creation by other means than cloning, but for now we concentrate on the normal situation.) An initial object and all objects recursively cloned from it during all executions of the program constitute a *clone family.* We write $\overline{\omega}_i$ for ω_i's clone family[†]. For instance, $\overline{\texttt{protoPoint}}$ denotes the set of all point objects cloned from an initial object `protoPoint`. Other objects such as `true` and `nil` are normally not cloned, so their clone families consist of the initial objects only, i.e., $\overline{\texttt{true}} = \{\texttt{true}\}$ and $\overline{\texttt{nil}} = \{\texttt{nil}\}$.

The next step towards defining types is to *group* the initial objects according to similarity. One may think of the groups as a *partition* of the image induced by an equivalence relation "is-similar-to." We let $G_1, G_2, \ldots, G_m$ denote the groups (here $m \le n$). Since the groups form a partition of the image, they are non-empty, pair-wise disjoint, and cover the image: $\bigcup_i G_i = IM$.

Section 4.6 describes exactly how groups are delimited. For now, it suffices to think of a group as a set of similar initial objects. For example, the 10,000 point objects may form a "point" group, all smallInts may form a "smallInt" group, and the floating point numbers may form a "float" group. Indeed, we will usually use suggestive names for groups rather than defining them precisely in terms of initial objects.

Above we closed initial objects under the "cloned-from" relation and obtained clone families. Now we close groups to obtain object types. For each group $G \subseteq IM$, the *object type* $\overline{G}$ is defined by:

[†] We apologize for the clash of notation with negation in boolean algebra. Since types occur frequently in this dissertation, there was a demand for a succinct notation. We settled on the bar since it reminds us of the closure operation involved in defining types.

$$\overline{G} = \bigcup_{\omega \in G} \overline{\omega}.$$

In other words, an object type is the union of a set of clone families, thus it is a set of objects. Since groups form a partition of the initial objects, object types form a partition of the set of all objects. Thus, any object that exists during any program execution belongs to (or "has") a unique object type. We use the bar consistently to mean closure under cloning: $\overline{\omega}$ is a single object closed under cloning (i.e., a clone family), whereas $\overline{G}$ is a group closed under cloning (i.e., an object type).

Object types are analogous to, but more general than, classes in a class-based language such as Smalltalk. In Smalltalk, each object is an instance of (or "has") exactly one class; in Self, each object belongs to (or "has") exactly one object type. In Smalltalk, knowing an object's class reveals the structure of the object (how it is laid out in memory, how many instance variables it has, etc.); in Self, knowing an object's object type reveals a set of (similar) possible structures, one for each initial object in the corresponding group. We are now, finally, ready to define what a type is:

Definition. A *concrete type* (or simply *type)* is a set of object types.

It is convenient to let U denote the universe of object types, i.e., $U = \{\overline{G}_1, \overline{G}_2, \ldots, \overline{G}_m\}$. Now a type is simply a subset of U. U itself is also a type. We define the extension of a type as the set of objects whose object type belongs to the type. More precisely, let $T \subseteq U$ be a type. Then

$$\text{extension}(T) = \bigcup_{\overline{G} \in T} \overline{G}.$$

In particular, extension(U) is the set of all objects that may exist during any execution of the program.

To summarize, a type is a set of object types, an object type is the closure under cloning of a group, and a group is a set of similar initial objects. Finally, the extension of a type is the set of all objects, whose object type is a member of the type. Using object types such as $\overline{\text{true}}$, $\overline{\text{false}}$, $\overline{\text{nil}}$, $\overline{\text{smallInt}}$, and $\overline{\text{bigInt}}$, with the obvious meanings, here are some examples of expressions and the types they may have.

- type(`x<y`) = $\{\overline{\text{true}}, \overline{\text{false}}\}$.

  Boolean expressions have a type with two members. This type reflects a *choice* to assign `true` and `false` to different groups (a possible justification may be that they implement `ifTrue:False:` differently).

- type(`nil`) = $\{\overline{\text{nil}}\}$.

  The type of the expression `nil`, which returns the object `nil`, is $\{\overline{\text{nil}}\}$. Usually, `nil` is not cloned, so the extension of the type $\{\overline{\text{nil}}\}$ consists of a single object: {`nil`}.

  Even if a language, like C++, does not treat `nil` (or `NULL`, to use C++ terminology) as an object, it may be worth doing so during type inference in order to win a complimentary "`NULL` analysis": types that include $\overline{\text{NULL}}$ identify where `NULL` pointers could show up during execution.

- type(`nil*3`) = {}.

  The empty type describes an expression that cannot produce a result. This can happen in several ways. For example, the expression may fail (such as the above one) or it may invoke a block method that performs a non-local return (i.e., it does not return to the caller). Some primitives, such as `_Restart` and `_Quit`, also have empty result types because they do not return to the caller.

- type(`141*143`) = $\{\overline{\text{smallInt}}, \overline{\text{bigInt}}\}$.

  The inferred types of arithmetic expressions often contain both $\overline{\text{smallInt}}$ and $\overline{\text{bigInt}}$. Unless the type-inference algorithms perform range analysis, arithmetic overflows cannot be ruled out.

We have defined what a type is, but still need to state precisely what it means for expressions and slots to "have" a type. However, we must first clarify what the terms "expression" and "slot" mean in the context of type inference. For type inference to be feasible, the number of expressions and slots to infer types for must be finite. In traditional programming languages and environments, such as Pascal, where programs are represented as source code (or an equivalent form like abstract syntax trees), the terms “expression” and “slot” have been given textual (syntactic) meanings: an expression or a slot *is* the piece of program text (or the syntax tree node) that defines it. Thus, a program defines a finite number of expressions and slots. For the Self type inferencer, we want expressions and slots to have similar meanings. However, with the Self environment’s de-emphasis on source code, we cannot characterize expressions and slots as textual or syntactic entities. Instead, we turn to objects.

Consider first expressions, which in Self are found in method bodies only. The image in which the target program is found contains only a finite number of methods. Thus, if methods cannot be cloned during execution, we can define expressions by reference to methods. If, however, methods can be cloned, an unlimited number of methods may exist over all program executions[†]. To limit the number of expressions to infer types for, we can elect not to distinguish between clones of a method during type inference, similar to the way we closed objects under cloning when defining object types. In either case, the type inferencer only needs to infer types for finitely many expressions: one for each expression in a method in the image *IM* of initial objects.

Now consider slots. Local slots, including arguments, which have scope within a single method, are handled like expressions: the type inferencer computes a type for each argument or local slot found in a method in *IM*. Thus, even if a method may be invoked many times in each execution of the target program, the type inferencer (to a first approximation) computes only one type for each of the method’s arguments and local slots. Figure 7 summarizes how types are assigned to expressions and local slots.

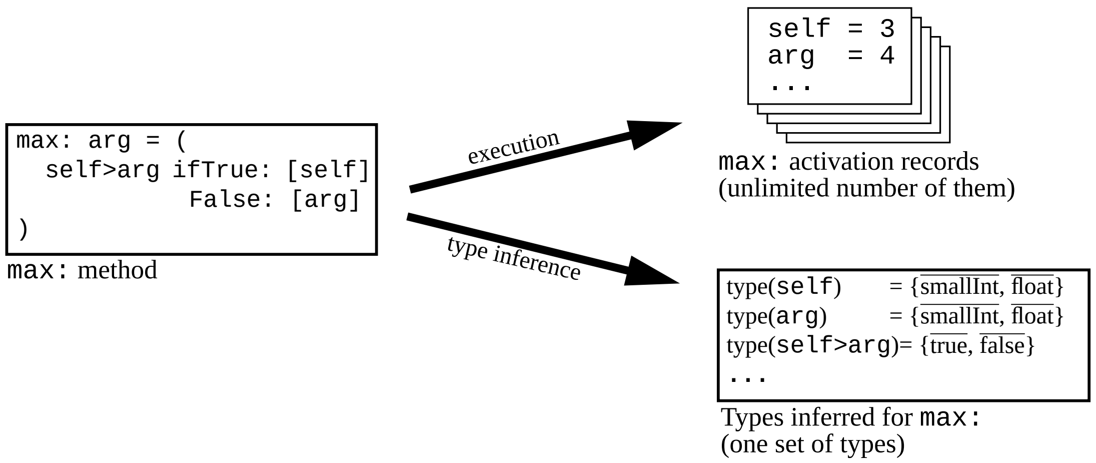


Figure 7. Assigning types to expressions and local variables

A method such as `max:` may be executed many times, giving rise to many activation records. The type inferencer computes a type for each expression and local slot in the method. These types are general enough to accommodate the objects in all the execution-time activation records.

For “instance slots,” i.e., those that are defined in objects instead of methods, finiteness also becomes a concern. Here, the problem is cloning: during execution objects may be cloned, resulting in an unlimited number of “run-time slots.” For type inference to be feasible, we must reduce this intractable number of run-time slots to a finite number of “type-inference-time slots.” Fortunately, we have already established a structure that can ensure finiteness: the object types. Instead of attempting the intractable feat of inferring a type for each slot in each run-time object, the type inferencer infers a type for each slot in each object type. Since there are only finitely many object types, there are only finitely many slots for which to infer types. For example, consider cartesian point objects. Instead of attempting to compute a type for each `x` slot in each point object existing in any execution of the target program, the inferencer computes a

[†] The reflective operations in Self *can* clone methods during program execution; however, it happens relatively infrequently.

single type for the `x` slot in the object type $\overline{\text{point}}$. Figure 8 illustrates the relationship among types of slots, object types, and objects.

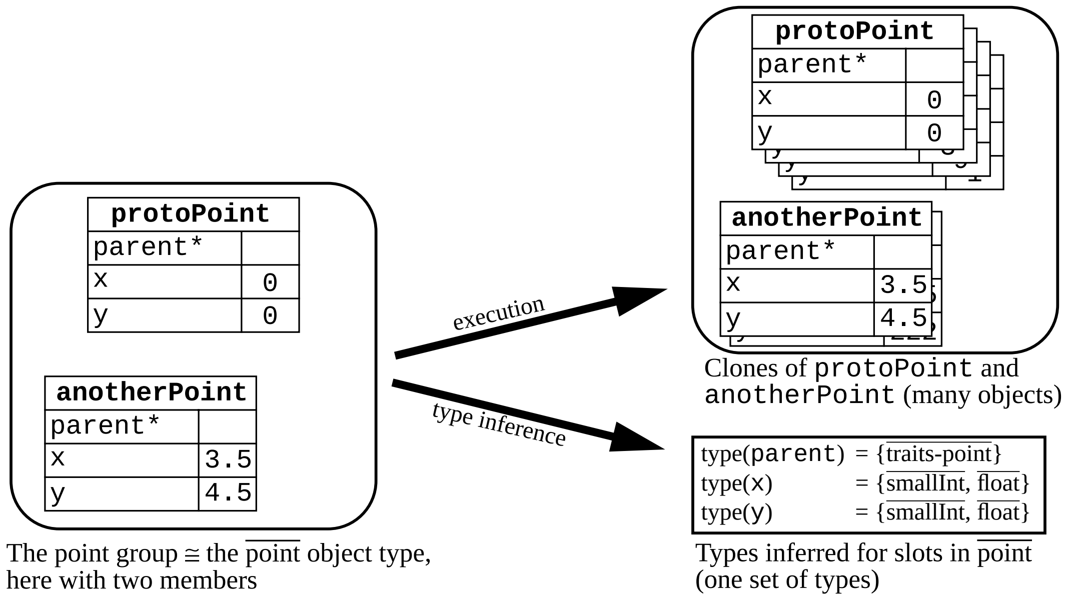

protoPoint
parent*
x 0
y 0
anotherPoint
parent*
x 3.5
y 4.5
The point group ≅ the point object type, here with two members
execution
type inference
protoPoint
parent*
x 0
y 0
anotherPoint
parent*
x 3.5
y 4.5
Clones of protoPoint and anotherPoint (many objects)
type(parent) = {traits-point}
type(x) = {smallInt, float}
type(y) = {smallInt, float}
Types inferred for slots in point (one set of types)

Figure 8. Assigning types to instance slots

An initial object such as `protoPoint` may be cloned many times during execution. To keep type inference feasible, the inferencer computes a type for each slot in each object type instead of attempting to infer a type for each slot in each object. These types must be general enough then to accommodate all the objects in the extension of the object type.

For the reader who is more familiar with Pascal-style textual environments than Self's object-based environment, thinking of types as being inferred for textual expressions and slots will provide adequate intuition for an in-depth understanding of the type-inference algorithms.

We can now define what it means for an expression to have a type. The definition quantifies over all possible executions of the target program to soundly approximate its behavior (recall that although our definitions of programs and program executions for simplicity assume that programs take no arguments, repeated executions of the same program can produce different computations by reading different external data). In addition, we quantify over all evaluations of the expression within each execution[†]:

- An expression *E has* type $T_E \subseteq U$ if for all executions of the target program, whenever *E* is evaluated, it yields an object that is in the extension of T_E.

  Equivalently, *E* never evaluates to an object outside extension(T_E).

Next, consider what it means for an instance slot to have a type. We quantify over program executions (since these may differ, as explained above), time (since assignable slots may contain different types of objects at different times)[‡], and all currently-live objects in the extension of the object type (since clones of objects may have different types of objects in their instance variables than the cloned object has)[††]:

[†] The polyvariant type-inference algorithms for analyzing code with parametric polymorphism change this quantifier to gain precision; see Section 3.2.

[‡] The algorithms for analyzing variable-accesses flow-sensitively change this quantifier to gain precision; see Section 4.5.

- A slot S in the object type $\overline{G}$ *has* type $T_S \subseteq U$, if for all executions of the target program, at any moment in time, the contents of the slot S in all objects with object type $\overline{G}$ is an object in the extension of T_S.

  Equivalently, no object in $\overline{G}$ ever contains an object in its S slot that is not in extension(T_S).

For brevity, we omit the definition for argument and local slots of methods. It falls directly along the lines followed above for expressions and instance slots.

The above definitions capture the usual intuition of what it means to have a type. For example, an expression that always returns `true` or `false` has the type {$\overline{\text{true}}$, $\overline{\text{false}}$}, whereas an expression that can return `true`, `false` or `47` cannot have this type. For another example, if the target program uses linked lists to hold smallInts and floats, the `contents` slot in the $\overline{\text{linked-list}}$ object has type {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}.

By definition, if an expression or slot has type T, it also has type T' for any T' that is a superset of T. This property permits conservative type inference: the inferred types may overestimate what can actually occur during execution. The universal type U is trivially sound for any slot or expression, but it is not a useful type to infer since it conveys no information (any object belongs to its extension). Indeed, the goal is to infer the most precise types, where a more precise type is one with a smaller extension.

Table 4 summarizes the definition of type and the notation used.

Concept	**Example(s)**	**General**
Image/initial objects	*IM* = {`nil`, `true`, `protoPoint`, `aPoint`, `47`, …}	$IM = \{\omega_1, \omega_2, \ldots, \omega_n\}$
Clone family	$\overline{\texttt{protoPoint}}$	$\overline{\omega_j}$

Group	point = {`protoPoint`, `aPoint`}
	colorPoint = {`aColorPoint`}
		$\varnothing \subset G_i \subseteq IM$

Object type	$\overline{\text{point}} = \overline{\texttt{protoPoint}} \cup \overline{\texttt{aPoint}}$ $\overline{\text{colorPoint}} = \overline{\texttt{aColorPoint}}$	$\overline{G_i} = \bigcup_{\omega \in G_i} \overline{\omega}$
Universe	U = {$\overline{\text{nil}}$, $\overline{\text{true}}$, $\overline{\text{false}}$, $\overline{\text{point}}$, $\overline{\text{smallInt}}$, …}	$U = \{\overline{G_1}, \overline{G_2}, \ldots, \overline{G_m}\}$
Type	{$\overline{\text{point}}$, $\overline{\text{colorPoint}}$}	$\varnothing \subset T \subseteq U$
Extension	extension({$\overline{\text{point}}$, $\overline{\text{colorPoint}}$}) = $\overline{\text{point}} \cup \overline{\text{colorPoint}}$ = $\overline{\texttt{protoPoint}} \cup \overline{\texttt{aPoint}} \cup \overline{\texttt{aColorPoint}}$	$\text{extension}(T) = \bigcup_{\overline{G} \in T} \overline{G}$

Table 4. Summary of the definition of type

3.1.3 Discussion

To better understand the definition of type, we discuss some of its properties and limitations below. We start with some of the consequences of grouping, then look at the effects of using types that are closed under cloning.

While we have not yet specified exactly how groups are delineated, grouping has consequences for the attainable precision of type inference. For example, the decision to consider all smallInts as belonging to one group, rather than assigning them to each their own (e.g., $\overline{46}$), implies that during type inference it is impossible to distinguish between different smallInt values. Clearly, this lack of distinction makes type inference more efficient, since a single analysis is valid for all smallInt values. However, sometimes it introduces inaccuracies that a value-based analysis could avoid. These inaccuracies may go beyond the precision loss initially incurred by the coarser grouping:

- The type inferred for

```
3 = 3 ifTrue: [6] False: [nil].
```

[††] The algorithms for analyzing code with data polymorphism modify the notion of object type (effectively changing this quantifier) to gain precision; see Section 4.3.

is {$\overline{\text{smallInt}}$, $\overline{\text{nil}}$} rather than {$\overline{\text{smallInt}}$} or even {$\overline{6}$}, since grouping abstracts the condition from `3 = 3` to $\overline{\text{smallInt}} = \overline{\text{smallInt}}$; thus, the type inferencer can no longer tell that the outcome of the test is always true, and therefore must include the type of the false branch, {$\overline{\text{nil}}$}, in the type of the conditional.

- Self's automatic coercion of smallInts to bigInts interacts with the grouping of all smallInts to cause precision loss: the type inferencer can no longer rule out overflows and, consequently, infers the type {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$} for the sum, product, difference, and quotient of two smallInts. While sound, this type is often more pessimistic than necessary; e.g., smallInt division can overflow only in a single case, created by the asymmetric negative and positive ranges of two's complement representations:

$$\frac{-2^{29}}{-1}\,.$$

There are ways to recover at least some of the lost precision, although we have not implemented such algorithms. As will become clear later, type inference builds a call graph of the analyzed program. Hence, after type inference, the standard data-flow machinery can be applied to compute many kinds of quantities, including value-based ones. Thus, it is possible to recover some of the finer resolution of a value-based analysis by following type inference with separate analyses such as constant propagation and constant folding. However, two separate analyses will in general be less precise than one combined, since the first of the two separate analyses is unable to take advantage of the information discovered by the second analysis. We discuss a generalization of this problem in Section 4.1.3.

To illustrate some of the flexibility permitted by our type system, consider boolean expressions. Above, we assigned the two boolean objects different object types, $\overline{\text{true}}$ and $\overline{\text{false}}$. Consequently, an expression that returns a boolean is polymorphic (i.e., it has more than one object type in its type). An alternative, and a perfectly reasonable one, is to define a group bool = {`true`, `false`}. Now, a boolean expression is no longer polymorphic. It has a type with a single member: {$\overline{\text{bool}}$}.

Object types are closed under cloning since they are unions of clone families. This property ensures finiteness of the domain of types, but also makes it impossible for the type inferencer to distinguish between two objects that are clones of each other. For example, if a rectangle with integer coordinates is cloned, and the clone's coordinates are changed to be floats, then it appears to the type inferencer as if both rectangles may have either integer or float coordinates (since the type inferencer only infers one type for the entire rectangle clone family). Consequently, code with data polymorphism cannot be analyzed with maximal precision. We will discuss possible solutions to this problem in Section 4.3. These solutions involve splitting object types into a finite number of "refined object types" that can have differently-typed instance variables.

3.2 Type-inference algorithms

Now that we have established the framework, we are ready to present several constraint-based type-inference algorithms. Table 5 lists the algorithms, the section that describes each one, and a reference for each one. The table also specifies which algorithms are "polyvariant" (defined in Section 3.2.2) and "adaptive" (defined in Section 3.2.5).

Algorithm	**Reference**	**Polyvariant?**	**Adaptive?**	**Section**
basic	[86]	no	no	3.2.1
1-level expansion	[86]	yes	no	3.2.3
p-level expansion	[92]	yes	no	3.2.4
hash function	[6]	yes	yes	3.2.6
iterative	[93]	yes	yes	3.2.7
cartesian product	[3]	yes	yes	3.2.8

Table 5. The six type-inference algorithms we analyze

We focus on how the algorithms analyze code with parametric polymorphism[†]. The algorithms are presented in order of increasing power and precision. This order coincides approximately with the order in which the algorithms were developed, with the possible exception of the iterative and the cartesian product algorithms that were developed simultaneously and independently.

Although the cartesian product algorithm presented in Section 3.2.8 is the main contribution of this chapter, the description and analysis of the other algorithms leading up to it are important as well:

- We introduce templates and use them to uniformly present and directly compare the algorithms.
- One algorithm's shortcomings can often be used to motivate the next more powerful algorithm.
- The sequence of ever more precise algorithms conveys an in-depth understanding of why polymorphic code is non-trivial to analyze.

The remainder of this chapter describes the six algorithms.

3.2.1 The basic algorithm

Palsberg and Schwartzbach presented the basic type-inference algorithm in 1991 [86]; later, Grove studied the algorithm's applicability to optimizing Cecil programs [51]. The basic algorithm has deficiencies when analyzing polymorphic code, but it constitutes the core of all the improved algorithms, so we review it in some detail. Furthermore, understanding its weaknesses is the key to developing better algorithms. Palsberg and Schwartzbach used a mathematical formalism, describing type inference as a constraint-solving problem:

- Derive a set of conditional constraints (subset relations) from the program being analyzed.
- Solve the constraints using a fixed-point algorithm.
- Map the solution back onto the program to obtain the desired type information.

We take a more operational view, presenting type inference as a combined control- and data-flow analysis over the abstract domain of types. This abstract interpretation perspective [35] allows a direct correspondence between analyzing a program and executing it, and, we hope, is closer to the operational understanding that most programmers have of programming language semantics.

We describe the basic algorithm below. Following the description, several subsections are devoted to extracting insights about not just the basic algorithm, but constraint-based analysis in general. Section 3.2.1.1 presents constraints for important and common language constructs, including assignment and method invocation. Section 3.2.1.2 defines "templates," an original contribution and an essential concept for uniformly describing and understanding all the improved type-inference algorithms. Section 3.2.1.3 establishes the foundation for an intuitive understanding of constraint-based analysis algorithms by outlining how analysis-time concepts correspond to well-known run-time concepts. Finally, Section 3.2.1.4 returns the focus to the basic algorithm, discussing its specific strengths and weaknesses.

The basic algorithm takes three steps to infer types (these steps do not correspond one-to-one to the steps in Palsberg and Schwartzbach's perspective on constraint-based analysis).

Step 1. Allocate type variables. Constraint-based analysis starts by allocating a *type variable* to every slot and expression in the target program. Figure 9 shows a program fragment and the corresponding type variables. A type variable is simply a variable whose possible values are types, i.e., sets. Initially, all type variables hold the empty type, but the next two steps add enough object types to them to make them hold sound types for the slots and expressions with which they are associated. Nothing is ever removed from a type variable. This *monotonicity* property is a integral part of the analysis algorithms. In particular, the precise resolution of dynamically-dispatched sends relies on

[†] Since we focus on parametric polymorphism, whenever it is clear from the context we will usually be brief and simply write "polymorphism."

it (see Section 4.1). Monotonicity can also be used to formally reason about complexity and termination properties of the algorithms, although we will generally refrain from doing so here.

For simplicity, we think of all type variables as being allocated in a single step. In practice, the type inferencer allocates the type variables for expressions and slots when it first encounters these during inference. This way, only the slots and expressions that the type inferencer deems may take part in the execution of the target program need to have type variables allocated.

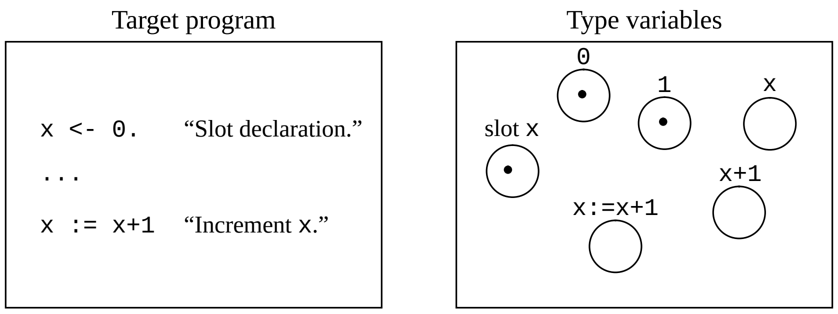


Figure 9. Programs and type variables

The first step of type inference associates a type variable with every slot and expression in the target program. The left frame shows a program fragment, and the right frame shows the type variables (in no particular order). The type variable for the *slot* `x` is labelled "slot `x`" to distinguish it from the type variable for the *expression* `x` that reads this slot.

Step 2. Seed type variables. The second step changes the type variables created in the first step. The goal is to capture the *initial state* of the target program, i.e., the state just prior to execution. The initial state is captured by initializing type variables that correspond to slots or expressions in the program where objects are found initially: the type variables are *seeded.* In Self, two cases require seeding:

- *Slots with initial values.* An initialized slot's type variable has the object type of the slot's contents added to it. For example, the type variable for the slot `x←nil` has $\overline{\text{nil}}$ added to it, so after seeding it holds the type $\{\overline{\text{nil}}\}$.
- *Literal object expressions*. Literal objects are expressions such as `1`, `3.14`, `'string'`, but also include blocks and regular objects defined "inline" in method bodies. For example, the type variable for the literal object `'bicycle'` has $\overline{\text{string}}$ added to it.

After step 2 is completed, some of the type variables will have a single member, and the rest will still be empty. Figure 9 shows a seeded type variable's member as a small dot. The seeding of type variables, like their allocation, is performed incrementally, in practice by seeding type variables immediately upon allocating them.

Step 3. Establish constraints and propagate. The previous step captured the initial state of the target program, setting the stage for the final step which will capture the *execution* of the target program. To capture the execution, the third step builds a global data-flow graph around the type variables that the previous steps allocated and seeded.

Specifically, the third step builds a directed graph whose nodes are the type variables. The edges, which are added one by one, represent *constraints.* A constraint is the type-inference-time equivalent of an execution-time data flow. For example, if the target program executes the assignment `x:=exp`, there is a data flow from `exp` to `x`. The data flow means that any object that can be the result of `exp` can also be in the `x` slot. Thus, to ensure soundness of the inferred types, any object type in the type of `exp` should also be in the type of `x`. When the algorithm encounters this data flow, it adds an edge from `exp`'s type variable to `x`'s type variable, reflecting that type(`exp`) ⊆ type(`x`). Figure 10 depicts this constraint.

Whenever a constraint (i.e., an edge) is added to the graph, object types are *propagated* along it. In Figure 10, all the object types in the `exp` node are pushed along the arrow to the `x` node. As more and more constraints are added to the

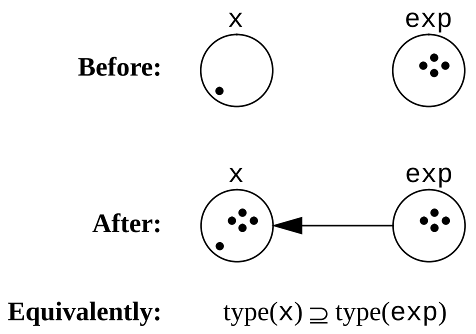


Figure 10. The effect of a constraint

The situation before and after establishing the constraint for `x:=exp`. Once the constraint is established, all the object types in `exp`'s type variable flow along the arrow to `x`'s type variable, establishing (and maintaining, since the constraint remains) that type(`x`) $\supseteq$ type(`exp`).

graph, the object types that were originally only in the seeded type variables can flow further and further. The eager propagation of object types along the edges ensures that subset relations such as type(`exp`) $\subseteq$ type(`x`) always hold: if an object type is added to type(`exp`), it is immediately propagated to type(`x`), reestablishing the subset relation.

Which constraints should be generated? One for *every* possible data flow in the target program; otherwise soundness of the inferred types cannot be ensured. Several concrete examples of constraints are given in Section 3.2.1.1. When a constraint is added, more propagation becomes possible simply because there is a new edge along which object types can flow. The reverse also holds: when propagation makes the receiver type of a send grow, dynamic dispatch means that the send may invoke new methods, hence more constraints may be needed to capture these invocations. To account for this interaction, step 3 consists of repeatedly establishing constraints and propagating, until no more can be done.

Type inference must terminate because there are only a finite number of type variables and object types. In the worst case, type inference terminates after propagating every object type into every type variable and creating a constraint between every pair of type variables. (Since object types are never removed from type variables, at this point no more can be done.) In this regard, type inference resembles algorithms for computing transitive closures of finite graphs: only so many edges can be added before lack of further progress forces termination.

When step 3 terminates, the type variables have become the nodes in a (large) directed graph. The type of an expression or slot in the target program can then be found by consulting the corresponding node in the network. The types are sound because:

- The initial state was correctly captured, i.e., the seeding of type variables accounted for all initial locations of objects.
- All possible executions were correctly captured, i.e., a constraint was established for every possible data flow, ensuring that if an object can flow from an initial location to some other place during an execution of the target program, the corresponding flow path exists in the constraint network.

This informal soundness argument demonstrates the power of thinking of constraint-based analysis as a flow problem and emphasizing the direct correspondence between analysis time and run time.

3.2.1.1 Constraints for common language constructs

We first consider the simplest possible statement, the assignment `x:=y`. The right hand side of the assignment is an expression `y` that reads a slot `y`. The assignment statement involves two data flows: one from the slot `y` to the expression y, and one from the expression y to the assigned slot x. Since there are two data flows, the resulting constraint graph has two edges, as shown in Figure 11.

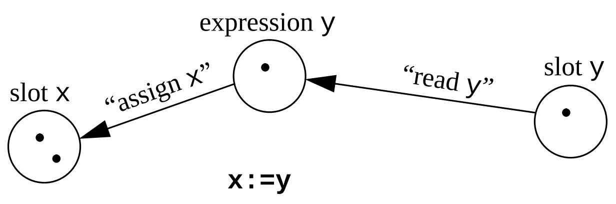


Figure 11. Constraints for an assignment

A simple assignment such as x:=y generates *two* constraints: one to read the value of the y slot and one to set the value of the x slot. For legibility, we have labeled the constraints with the data flows they represent.

Different languages will have different constructs that produce data flows. Some common examples are:

- An assignment generates a data flow from the new value expression to the assigned variable (Figures 10 and 11).
- A variable read generates a data flow from the accessed variable to the reading expression (Figure 11).
- A method invocation (or function call) generates data flows from the actual argument expressions to the formal arguments of the invoked method (we consider the receiver expression of a message send an actual argument, and the self slot of the invoked method a formal argument). Furthermore, a data flow returns the result of the invoked method to the send invoking it. All these constraints are illustrated in Figure 12 for a method with a single argument.
- Primitive data types (booleans, smallInts, and floats) and their operations, including built-in control structures, generate data flows. Figure 13 shows the constraint for Self's _Clone primitive. It is particularly simple because the type system does not distinguish between an object and clones of it. Figure 14 shows the constraints for a conditional expression such as Scheme's if-then-else[†]. By merging the results of the conditional's two branches, the constraints reflect the conservative assumption that both branches may be taken.

More examples of constraints are found in [6, 85, 86, 93].

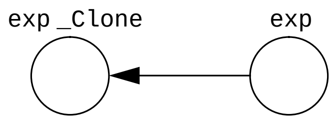


Figure 13. Constraint for the _Clone primitive

Self primitives generate data flows. For the _Clone primitive, the resulting constraint is from the expression computing the object to be cloned to the clone expression as a whole. The constraint ensures that the two types are the same, reflecting that the type-inference algorithms do not distinguish between members of the same clone family.

In Self, essentially all expressions are dynamically-dispatched message sends. Messages are used to read variables, to assign variables, and to invoke methods (of course). From the syntax of a send, it is not possible to tell which of these

[†] Self defines conditional statements using objects and dynamic dispatch. We will later see how the rules for message sends translate Self's conditional statements into constraint structures similar to the one describing Scheme's built-in conditional expression.

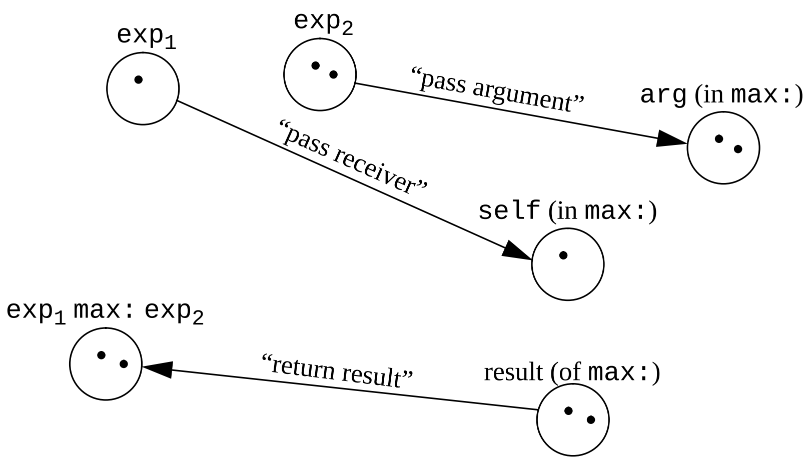


Figure 12. Constraints for a method invocation

A method invocation, here illustrated by a send `exp`$_1$ `max: exp`$_2$ invoking a method called `max:`, generates constraints for passing the receiver expression into the `self` slot of the invoked method, for transferring arguments (here, just one), and for returning the result of the method back to the send invoking it. The constraints generated by the expressions in the body of the invoked method are not shown here in order to keep the picture simple.

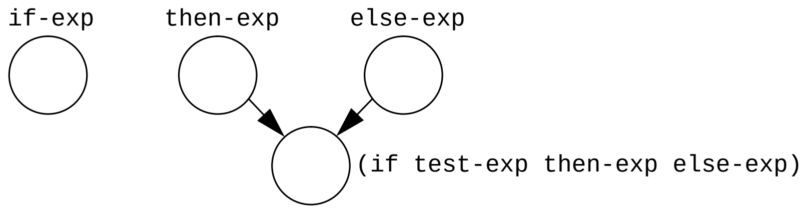


Figure 14. Constraints for Scheme's `if` expression

The *value* of a Scheme `if` expression is either the value of `then-exp` or `else-exp`. In terms of constraints: the *type* of a Scheme `if` expression is the union of the types of `then-exp` and `else-exp`.

effects it may have. However, given the assumption that a send invokes a particular method, the constraints that are needed to capture the effect of the method invocation are the same, whether the send is statically or dynamically bound to the method. Similarly, given the assumption that a statement assigns to a variable, the constraints that describe the assignment do not depend on whether the assignment statement is a "traditional" statically-bound assignment statement or a dynamically-dispatched send. This observation is important because it allows separation of two problems:

- Determining the possible targets of a dynamically-dispatched send.
- Analyzing the targets, once they have been found.

Above, we showed how constraint-based analysis solves the second of these problems. We defer the description of how the first problem, dispatch resolution, is solved until Section 4.1. For now, it is sufficient to know that there is *some* way to find a precise (and conservative) approximation to the set of targets a given send may invoke. When the dispatch resolution predicts more than one possible target, constraints are generated for all the possible targets, following the common conservative assumption that all predicted cases may occur during execution.

3.2.1.2 Templates

Constraint-based analysis builds a large network of type variables that are connected by constraints. Since even a medium-sized program may yield a network of thousands of type variables and constraints, this level of detail can be difficult to work with. Comparing two type-inference algorithms by looking at the networks they build is like comparing two cups of coffee by looking at individual atoms. The overall picture is lost in too many details and lack of structure.

To improve our understanding of constraint-based analysis and to facilitate comparisons of algorithms, the abstraction level must be raised above that of individual constraints. To accomplish this, we have formulated the concept of *templates*. They impose a structure on the constraint network and raise the abstraction level by encapsulating constraints, just as methods raise the abstraction level of programming by encapsulating expressions. Templates enable a concise exposition of the basic type-inference algorithm and the five different improvements we consider in Section 3.2.3 onwards.

Starting with the unstructured constraint graph produced by the basic type-inference algorithm, we can carve it up into a number of subgraphs, each corresponding to a method in the target program. These subgraphs are templates. More precisely, the template for a method *M* is the subgraph consisting of:

- The nodes (type variables) corresponding to expressions, local variables, formal arguments, and the return value† of *M*.
- The edges (constraints) originating from these nodes.

Thus, a template is the network that the basic algorithm generates from a single method. Nodes that correspond to instance variables are not part of any template, since instance variables do not belong to any particular method (they still have a type, of course).

For a first example of a template, consider the `max:` method found at the top of the number hierarchy in the Self system (for now it is safe to ignore that the method contains blocks; we discuss them in Section 4.4):

```
max: arg = ( self>arg ifTrue: [self] False: [arg] ).
```

Figure 15 shows the template for this method. The italicized text in the figure explains the different parts. The template is a box with two "input type variables" at the top, corresponding to the formal arguments `self` and `arg`, and a type variable for the result at the bottom. The interior structure of the template reflects the body of `max:`. Type inference has determined that the result of the comparison `self>arg` has type {$\overline{\text{false}}$, $\overline{\text{true}}$}, so contributions from two `ifTrue:False:` templates combine to form the result type of the method (remember: a conditional statement in Self is a message send of `ifTrue:False:` to `false` or `true`, hence the type inferencer connects conditional statements to templates for the `ifTrue:False:` methods in `false` and `true`, respectively). In the figure, the left-most `ifTrue:False:` template corresponds to the `ifTrue:False:` method in `false` (it selects the second argument) and the right-most `ifTrue:False:` template corresponds to the `ifTrue:False:` method in `true` (it selects the first argument). To simplify the picture, we have omitted the constraints for the comparison `self>arg` and the constraints that propagate its result into the `self` slots of the two `ifTrue:False:` templates.

Figure 15 also demonstrates how a send, `3 max: 4`, is connected to the `max:` template. The inference algorithm determines the send's type by propagating its actual argument types through the template(s) of the method(s) that the send may invoke. When a send may invoke several methods, its type is the union of the result types of these methods.

The example given in Figure 15 is very simple. To determine the type of the send expression, the inferencer can propagate the types through the template in a single pass from the top to the bottom. In general, the situation can be more complex since the constraint graph may contain cycles (e.g., caused by recursive methods). Thus, even though we

† In Self, the last expression in a method determines the returned value, so there is actually no need for a separate type variable for the return value. We will often draw it anyway to make clear what is returned. The return value also plays a special role when analyzing blocks with non-local return: see Section 4.4.1.

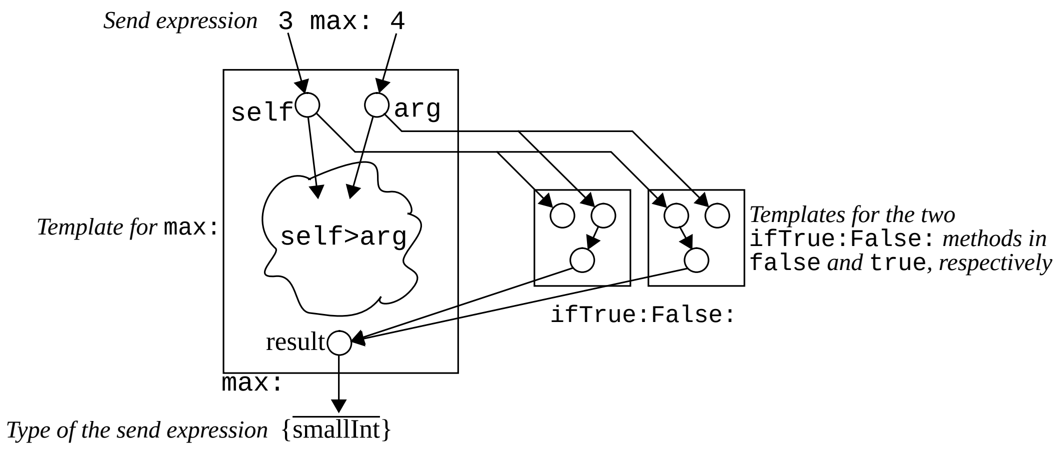

Send expression
3 max: 4
self
arg
self>arg
Template for max:
Templates for the two ifTrue:False: methods in false and true, respectively
ifTrue:False:
result
max:
Type of the send expression {$\overline{\text{smallInt}}$}

Figure 15. A template for the `max:` method

The type of the send expression is determined by propagating the types of its actual arguments through the template to find its result type.

show templates for simple methods to illustrate the various algorithms, the reader should keep in mind that the relation between input and output types of templates need not be straightforward.

Throughout the remainder of this chapter, we will work at the template level to avoid the complexity of dealing with large numbers of individual constraints. As will become clear, this level of abstraction is exactly right when comparing how the various inference algorithms analyze code with parametric polymorphism.

3.2.1.3 Relation between run time and type-inference time

It is helpful to understand constraint-based analysis by drawing parallels to execution of programs. The concepts describing constraint-based analysis correspond directly to familiar concepts describing program executions. Recall that object types are the analysis-time representation of objects, and constraints are the analysis-time representation of data flows. Extending this correspondence, templates represent activation records. Templates are created to analyze methods, whereas activation records are created to evaluate methods. Furthermore, *types* of formal argument slots and local slots can be found by inspecting type variables in templates, whereas during execution the *contents* of these slots can be found by inspecting fields in activation records.

For now we will be satisfied that a single template represents all possible activation records for the given method (see Figure 16). However, the more advanced type-inference algorithms may create several templates for each method. Any given activation record will still be represented by one template, and each template will in general still represent many activation records (the many-to-one correspondence is necessary since an unbounded number of activation records may be created during executions, whereas only a finite number of templates can be created during type inference). This is why we use the word “template”: they are templates for activation records.

Table 6 summarizes the relationship between run-time and type-inference-time concepts. We will continuously refer to this correspondence to support our intuition when describing the different improvements of the basic type-inference algorithm. However, before describing the improved algorithms, let us return briefly to the basic algorithm to better understand its strengths and weaknesses.

3.2.1.4 Discussion of the basic algorithm

Figure 17 illustrates the basic algorithm performing at its best: two different sends, each with an integer receiver and argument are connected to the template for `max:`. By propagating the object types through the network, the type-inference algorithm can establish that the return type of `max:` is {$\overline{\text{smallInt}}$}, i.e., that both sends have the type {$\overline{\text{smallInt}}$}.

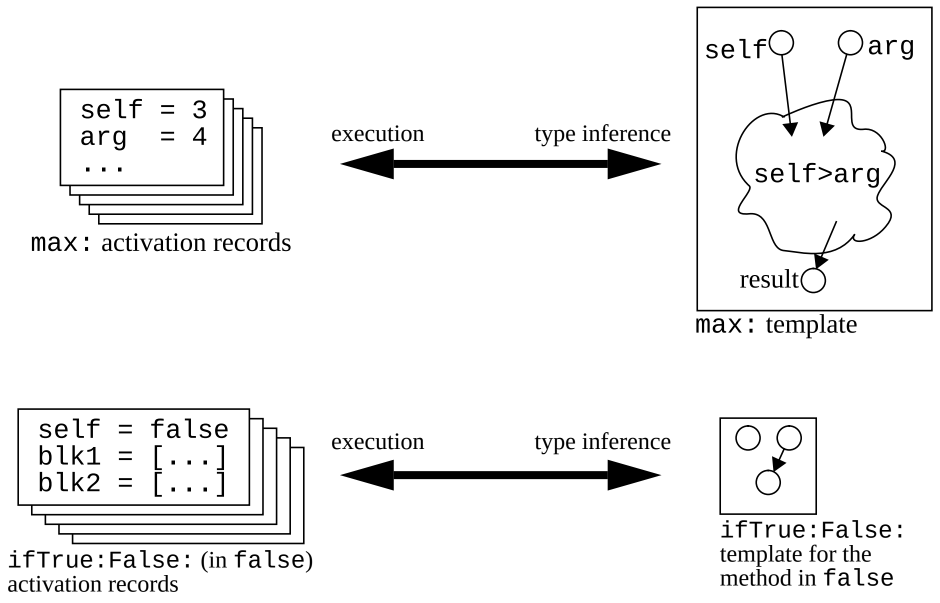


Figure 16. Relationship between activation records and templates

The basic algorithm creates one template per method, i.e., it establishes a many-to-one relationship between activation records and templates. More advanced type-inference algorithms may create several templates for some methods.

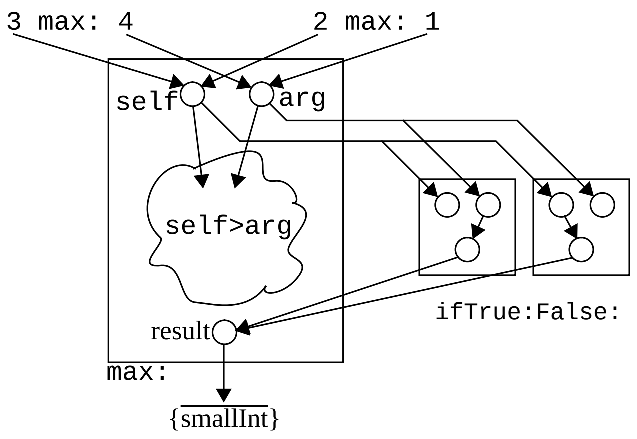


Figure 17. The basic algorithm when it succeeds

By propagating the $\overline{\text{smallInt}}$ object type from the inputs to the output, the basic type-inference algorithm can — optimally — determine that `max:` returns a smallInt.

The basic algorithm works well when all uses of a given method are similar. In particular, if there is no polymorphism in the target program, as in Figure 17, accurate types can be inferred. The algorithm also works well in some polymorphic situations. If both `max:` sends in Figure 17 instead had actual arguments of type {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}, propagation through the template would again determine the precise type for the sends: {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}. In summary, the basic algorithm excels when all sends invoking a method supply actual arguments of the same type.

The basic algorithm fails to infer precise types when two or more send expressions invoke the same method but supply different types of actual arguments. This loss of precision happens even if each send has simple monomorphic

Run time	Type-inference time
object	object type
activation record	template
contents of slot	type of slot
result of expression	type of expression
data flow	constraint

Table 6. Correspondence between run-time and type-inference-time concepts

actual arguments. For example, if one send invokes `max:` with smallInts, and another invokes it with floats, the types from these two monomorphic sends are mixed together so that both will have the imprecise type $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$ inferred; see Figure 18. The problem is that the basic algorithm creates only one `max:` template. Consequently, the types in the single `max:` template have to be general (i.e., large or imprecise) enough to cover all uses. Equivalently, the types in the `max:` template must be general enough to represent all `max:` activation records. The improved type-inference algorithms try to avoid this problem by creating several templates from each method.

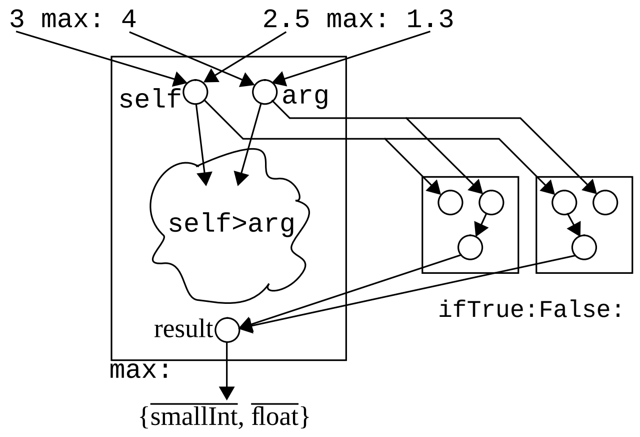


Figure 18. The basic algorithm when it fails

Under the basic algorithm, each method has only a single template and thus only a single result type. Therefore, the algorithm infers the type $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$ for both `max:` sends above, even though one computes the maximum of two integers and the other the maximum of two floats.

Is it possible that the basic algorithm works reasonably well on real programs, but just not on examples that are constructed or selected to show its weaknesses? The answer, unfortunately, is “no.” The mixing of types and subsequent imprecision of inference occurs extremely often. Self’s conditional statements constitute a particularly striking demonstration of this problem with the basic algorithm. As we have previously mentioned, conditional statements are implemented by a pair of methods with selector `ifTrue:False:` in the `true` and `false` objects respectively:

```
true = (|
  ifTrue: blk1 False: blk2 = (blk1 value). “Invoke true block.”
  ...
|).

false = (|
  ifTrue: blk1 False: blk2 = (blk2 value). “Invoke false block.”
  ...
|).
```

Even code this simple cannot be analyzed with a useful degree of precision. Specifically, if the algorithm infers that some particular use of `ifTrue:False:` may return a horse object, suddenly *all* uses of `ifTrue:False:` start returning horses! Figure 19 shows the constraint network corresponding to the following two conditional statements:

```
b1 ifTrue: [horse] False: [donkey]. “conditional 1”
b2 ifTrue: [tiger] False: [jaguar]. “conditional 2”
```

`b1` and `b2` are boolean receiver expressions with type {$\overline{\text{false}}$, $\overline{\text{true}}$}. Each conditional connects to two templates: the template for the `ifTrue:False:` method in `true`, and the template for the `ifTrue:False:` method in `false`. The type of each conditional, obtained by combining the contributions from the templates it connects to, is the disappointingly weak {$\overline{\text{horse}}$, $\overline{\text{donkey}}$, $\overline{\text{tiger}}$, $\overline{\text{jaguar}}$}. In general, `ifTrue:False:` will accumulate every type returned in any conditional statement in the target program.

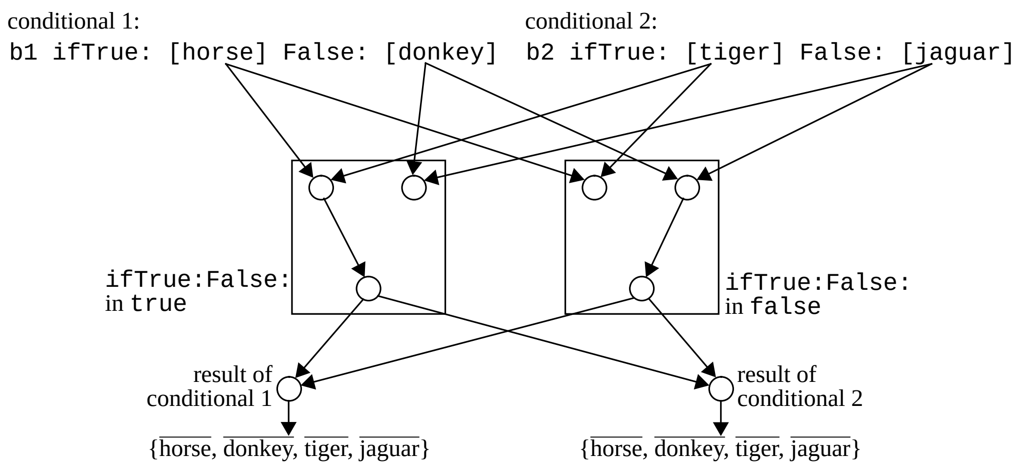

conditional 1:
b1 ifTrue: [horse] False: [donkey]
conditional 2:
b2 ifTrue: [tiger] False: [jaguar]
ifTrue:False: in true
ifTrue:False: in false
result of conditional 1
result of conditional 2
{$\overline{\text{horse}}$, $\overline{\text{donkey}}$, $\overline{\text{tiger}}$, $\overline{\text{jaguar}}$}
{$\overline{\text{horse}}$, $\overline{\text{donkey}}$, $\overline{\text{tiger}}$, $\overline{\text{jaguar}}$}

Figure 19. The basic algorithm applied to conditional statements

The basic type-inference algorithm does not keep different uses of conditional statements distinct. As a result, the inferred types are imprecise, e.g., the left-most conditional statement can never return a tiger, but the basic algorithm cannot rule it out.

We have illustrated the shortcomings of the basic algorithm with a specific example from Self. However, the situation that defeats the algorithm is so simple that analogous structures occur frequently in almost any real program, no matter what language it is written in. The `car` and `cdr` functions in LISP, the `new` method in Smalltalk, and iterators on user-defined data structures in any language are prime examples.

3.2.2 Polyvariant algorithms

The basic algorithm fails because it creates only one template per method. Consequently, each method is limited to having only one result type that must then be general (or imprecise) enough to accommodate all uses of the method. An algorithm, like the basic one, that analyzes each method once is called *monovariant*. The improved algorithms, presented in the following subsections, are all *polyvariant:* they may analyze methods multiple times in an attempt to improve precision. In terms of templates, monovariant algorithms create one template per method, whereas polyvariant algorithms may create multiple templates for some or all methods.

The distinction between monovariant and polyvariant analysis applies beyond constraint-based type inference. For example, Consel describes a polyvariant binding-time analysis in [32] (binding-time analysis determines which expressions a partial evaluator can compute at compile time). Consel parameterized his analyzer by a function that determines the degree of polyvariance. In this regard it is similar to the hash function algorithm presented in Section 3.2.6.

3.2.3 The 1-level expansion algorithm

Oxhøj, Palsberg, and Schwartzbach acknowledged that high-precision type inference requires avoiding the mixing of types that the basic algorithm suffers from [85, 86]. To this end, they proposed retyping each method for every syntactic send invoking it. For instance, if two different sends may invoke the `max:` method, types are inferred for it twice. This improvement enabled them to type small test programs that had previously proven too hard for a different type-inference algorithm developed by Hense [53].

We present their idea by the creation of several templates from each method in the program. Specifically, each send that may invoke a given method *M* is connected to its own *M*-template. Figure 20 illustrates this idea for two different sends that invoke `max:`. Now, instead of having a single *M*-template that corresponds to all *M*-activation records, there are several *M*-templates, each corresponding to the *M*-activation records that some particular send expression generates.

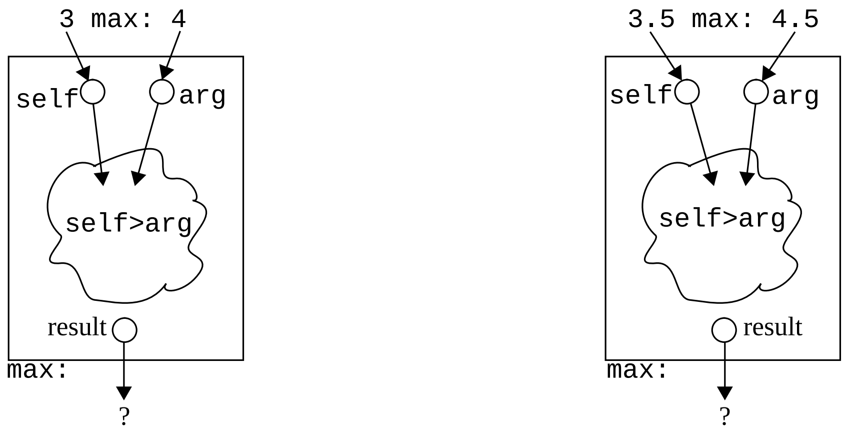


Figure 20. The idea behind the 1-level expansion algorithm

The 1-level expansion algorithm never shares a template between different send expressions.

Oxhøj, Palsberg, and Schwartzbach presented this improvement as a semantics-preserving source-to-source transformation (expansion) of the target program, followed by an application of the basic algorithm to the transformed program. Assume the target program contains *n* message sends, S_1, S_2, …, S_n, with some selector *sel*, and *k* methods, M_1, M_2, …, M_k, with the same selector. The expansion creates *n* copies of each method M_j: M_{j1}, M_{j2}, …, M_{jn}. The *i*'th copy, M_{ji}, is reserved exclusively for use by the *i*'th send S_i. (Of course, in most programs, every *sel*-send will not invoke every *sel*-method, in which case some of these copies are never used.) We call the resulting algorithm the "1-level expansion algorithm," because the expansion of the target program is done only once. (This algorithm is a special case of the "*p*-level expansion algorithm" described in Section 3.2.4.)

Expansion during type inference has much in common with inlining during compilation. The benefits are similar:

- Expansion allows a method to be typed in a specific context, hopefully improving the inferred types.
- Inlining allows a method to be compiled in a specific context, hopefully improving the generated code (elimination of the call overhead is often less significant than other optimizations made possible by compiling in a more specific context [59]).

The drawbacks of expansion and inlining are also analogous:

- Expansion increases the number of type variables and slows down type inference since the inferencer must analyze more methods.

- Inlining increases the compiled-code space and slows down compilation since the compiler must generate more compiled methods (each inlined method serves only one “caller”).

3.2.3.1 Discussion of the 1-level expansion algorithm

Below, we show that the 1-level expansion offers only a slim precision improvement over the basic algorithm, but first we give an example where the expansion actually helps. The 1-level expansion algorithm can precisely type these conditionals:

```
b1 ifTrue: [horse] False: [donkey].
b2 ifTrue: [tiger] False: [jaguar].
```

Figure 21 shows how the 1-level expansion algorithm performs better in this case than the basic algorithm: the fatal mixing of predators and prey is avoided because the 1-level expansion algorithm connects the two conditional statements to separate pairs of `ifTrue:False:` templates.

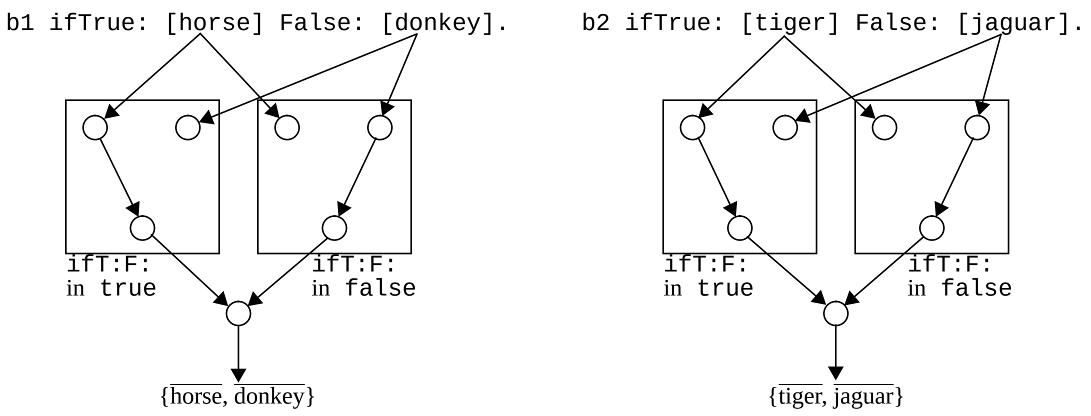


Figure 21. The 1-level expansion algorithm succeeds on conditional statements

The 1-level expansion algorithm avoids mixing different uses of `ifTrue:False:` by connecting different sends to distinct templates.

The predator/prey example shows that the 1-level expansion algorithm can be more precise than the basic algorithm. Unfortunately, the improvement is not spectacular. For example, the 1-level expansion algorithm still cannot infer precise types for two send expressions computing maxima of integers and floats, respectively. In fact, its precision in this case is no better than the basic algorithm’s. Figure 22 shows what goes wrong (compare with Figure 18). The 1-level expansion algorithm creates two `max:` templates but connects the `ifTrue:False:` sends inside these to the same pair of `ifTrue:False:` templates (since there is only one syntactic `ifTrue:False:` send in `max:` the expansion has no effect here). Thus, all that the 1-level expansion algorithm achieves over the basic algorithm is a delay in the mixing of types. With the basic algorithm the mixing happens immediately; with the 1-level expansion algorithm it happens in a pair of `ifTrue:False:` templates that the algorithm regrettably shares between the two `max:` templates it worked hard to keep separate in the first place.

The general situation exposed by the `max:` example is that of a multi-level polymorphic call chain. First, `max:` may be invoked on either integers or floats. Then, from within `max:`, the `ifTrue:False:` methods in `false` and `true` may be invoked with blocks that return integers or floats. Since there is only a single send of `ifTrue:False:` in the target program, the 1-level expansion algorithm creates only one pair of `ifTrue:False:` templates. The algorithm propagates both the integer- and float-blocks into this pair of templates and the types mix.

Perhaps the 1-level expansion algorithm works reasonably well on real programs, but just not on examples that are constructed to show its weakness? Again, the answer is “no.” Polymorphic call chains of depth greater than one are

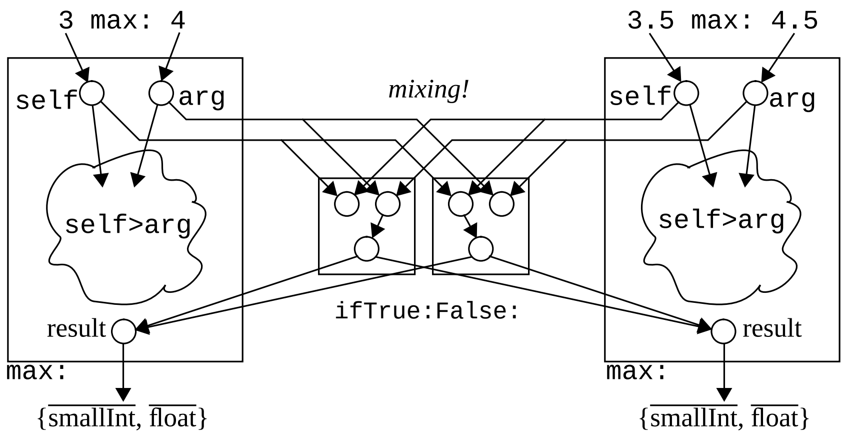

3 max: 4
3.5 max: 4.5
self
arg
mixing!
self
arg
self>arg
self>arg
ifTrue:False:
result
result
max:
max:
{$\overline{\text{smallInt}}$, $\overline{\text{float}}$}
{$\overline{\text{smallInt}}$, $\overline{\text{float}}$}

Figure 22. The 1-level expansion algorithm when it fails (`max:` method)

The 1-level expansion algorithm creates separate `max:` templates for two different sends that invoke `max:`. Unfortunately, the types still mix, since the two `max:` templates share a pair of `ifTrue:False:` templates.

common in real programs: all it takes to create such a chain is one method receiving a polymorphic argument and passing it to another method. The `max:` method demonstrated this situation, but in the Self system the most compelling example is perhaps the method `ifTrue:`. Found in the object `traits boolean`, this method is inherited by `true` and `false` and implements a one-armed conditional statement by sending `ifTrue:False:` to `self` (which is `true` or `false`):

```
    ifTrue: blk = ( self ifTrue: blk False: [] ).
```

Since `ifTrue:` is polymorphic, the single send in its body is polymorphic. Hence, every time `ifTrue:` is used in a Self program, the 1-level expansion algorithm will fail to keep the particular type distinct from other uses of `ifTrue:`. Even if a language has built-in control structures, polymorphic call chains that are more than one call deep remain common. Take quicksort: an implementation will likely consist of a polymorphic routine, `quickSort` that invokes another polymorphic routine, `partition`. A two level polymorphic call chain results.

In addition to imprecision, the 1-level expansion algorithm suffers from another problem: inefficiency caused by redundant analysis. Different sends with the same selector often supply the same types of argument, e.g., most sends of `+` add smallInts. The 1-level expansion algorithm does not recognize this pattern, and unnecessarily re-analyzes the smallInt addition method for each such send. (In Self the smallInt addition method is nontrivial since it handles type coercions and overflows into bigInts; see page 12.) Even a medium-sized program may add smallInts in hundreds of different places, and the cost of analyzing the smallInt addition method every time, just to discover again and again that the sum of two smallInts is a smallInt or bigInt, can quickly add up.

Palsberg and Schwartzbach computed a theoretical upper bound on the cost of the 1-level expansion algorithm relative to the basic algorithm: the expansion will in the worst case square the size of the analyzed program [86]. Phillips and Shepard (see Section 2.3.2.10) observed the high cost of such expansions in practice [92]. In their system, the basic algorithm took 30 minutes to infer types for the arithmetic expression `3+4` in ParcPlace Smalltalk (including the bigInt implementation). Applying a 1-level expansion (and at the same time introducing an orthogonal expansion to improve the analysis of data polymorphism†), the time to infer types for the `3+4` example increased to over 10 hours. Section 3.4 gives similar measurements obtained on the Self system. The Self execution times are much lower absolutely, and also indicate a lower relative penalty of each expansion level. However, the cost of each expansion level remains high: often a doubling in type inference time.

† Unfortunately, their report does not separate the effects of these two expansions.

3.2.4 The *p*-level expansion algorithm

Figure 22 illustrated how the 1-level expansion algorithm fails to analyze the `max:` method precisely because it must share a pair of `ifTrue:False:` templates. The obvious way to improve precision is to avoid the sharing, e.g., by using a *2-level expansion.* It is now, finally, possible to infer types precisely for `max:`, as Figure 23 demonstrates.

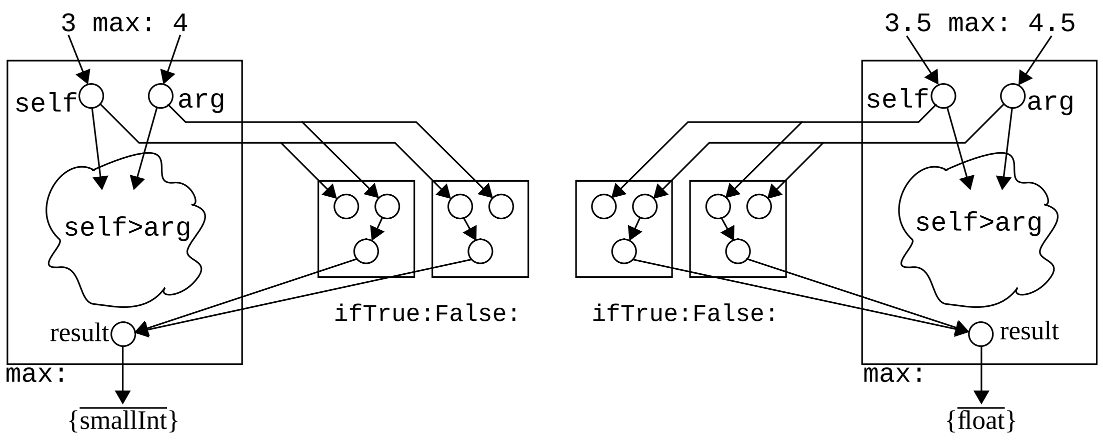


Figure 23. The 2-level expansion algorithm when it succeeds (`max:` method)

The 2-level expansion algorithm can analyze two differently-typed invocations of `max:` without mixing up their types.

The generalization of the 1- and 2-level expansion algorithms is the *p*-level expansion algorithm. In terms of Palsberg and Schwartzbach's program transformation, it can be described as follows:

Repeat the expansion *p* times, then apply the basic algorithm to the resulting program.

This idea has been proposed and even tried out several times. In the data flow community it is known as the "call string" approach to interprocedural data-flow analysis [105]: two calls, C_1 and C_2, can share an analysis of the method they invoke, if the last *p* calls on the paths leading to C_1 and C_2 are the same. Vitek Horspool, and Uhl also used call strings in a theoretical study of an analysis system for a Smalltalk-like language [120]; Palsberg and Schwartzbach applied it to the lambda calculus [87]; Phillips and Shepard implemented it in their type-inference system for Smalltalk [92].

3.2.4.1 Discussion of the *p*-level expansion algorithm

The *p*-level expansion algorithm is theoretically attractive because it offers a parameterized cost/precision trade-off. Precision can be turned up simply by increasing *p*. Unfortunately, for a given program, it is difficult to know in advance how many expansions are needed. For example, while a 2-level expansion is sufficient to analyze `max:` precisely, the following method that invokes `max:` twice to compute the maximum of three numbers, the receiver, `x` and `y`, needs a 3-level expansion:

```
max: x Max: y = ( (self max: x) max: y ).
```

A 3-level expansion is necessary because `max:Max:` produces a polymorphic call chain that is three calls deep: `max:Max:` → `max:` → `ifTrue:False:`. In general, a *p*-level expansion is precise as long as the deepest call chains that must be kept separate are at most *p* calls deep. Once the depth of calls exceeds the expansion level, precision is lost.

It turns out that a fairly high value of *p* is required to analyze many Self programs precisely. The explanation is that common code sequences "eat up" several expansion levels before the analyzer enters the user's code. For instance, the following family of messages is found in `defaultBehavior`, a common ancestor for almost any object:

```
defaultBehavior = (|
  value: a With: b With: c With: d = (self value: a With: b With: c).
  value: a With: b With: c         = (self value: a With: b).
  value: a With: b                 = (self value: a).
  value: a                         = (self value)
  value                            = (self).
|).
```

These methods enable blocks to strip unwanted arguments, and many places in the system depend on them. For example, if a 0-argument block is sent `value:With:With:With:`, the following "cascading" sequence of calls will strip the extra arguments one by one, before finally invoking the block method: `value:With:With:With:` $\rightarrow$ `value:With:With:` $\rightarrow$ `value:With:` $\rightarrow$ `value:`$\rightarrow$ `value`. Unfortunately, while these methods are convenient for the programmer, they are detrimental to the *p*-level expansion algorithm. When the block in the example is finally invoked, the argument-stripping call sequence has eaten up four expansion levels, leaving only $p - 4$ levels to handle the code in the block method.†

Based on this example, the *p*-level expansion algorithm is not very attractive for analyzing typical Self programs‡. Even worse, each expansion multiplies the redundancy problems that the 1-level expansion algorithm suffers from. Indeed, the theoretical worst-case cost of the *p*-level expansion algorithm is exponential in *p*, since the cost of a single expansion is quadratic.

3.2.5 Adaptive algorithms

Precise type inference must avoid mixing types when mapping run time to type-inference time. Types mix when two incompatible activation records are represented by the same template. This observation suggests creating lots of templates from each method so that each template needs to represent fewer activation records, hopefully lowering the chance of combining activation records that should preferably be kept separate. *Efficient* type inference, on the other hand, requires processing as few templates as possible, since every template created carries a computational cost.

We observed this dilemma with the 1-level expansion algorithm which, sadly, delivered the *worst* of both worlds: it did not create enough templates so types were still mixing, yet it created so many templates that performance suffered under redundant analysis.

To achieve the *best* of both worlds, a good type-inference algorithm must strike a fine balance between creating too few templates (resulting in low precision) and too many templates (resulting in slow type inference). Adaptive type-inference algorithms strive to achieve this optimum by creating many templates when it appears necessary, and sharing templates when it seems safe. A good adaptive algorithm should map similar activation records to a shared template and map disparate activation records to distinct templates. If successful, the result is precise type inference at a cost proportional to the "amount" of polymorphism in the target program.

3.2.5.1 The critical situation in adaptive type inference

Before describing specific algorithms, let us make the critical situation clear. Consider the send

† Throwing away arguments may seem weird. Why pass them in the first place? We chose the example for its conciseness. It is, however, easy to find other examples that do not rely on this particular style of programming. Indeed, the "opposite" situation, successively supplying default arguments by forwarding calls, also yields long polymorphic call chains. Consider this chain: `display` $\rightarrow$ `displayOn:` $\rightarrow$ `displayOn:InColor:` $\rightarrow$ `displayOn:InColor:Scale:`. To display an object on the default display, in the default color and scale, the object can be sent `display`. This method is implemented by a call to the more general method `displayOn:`, etc.

‡ The implementation of Self is highly optimized; often the compiler can inline calls several levels deep. Thus, long call chains resulting from highly factored code need not slow down execution. By making abstractions cheaper or even free, it is plausible that the Self compiler has influenced the programming style and caused an increase in the frequency and depth of polymorphic call chains in Self programs over other programs. Deep call chains can occur for other reasons as well. The "Law of Demeter," which was proposed to improve software maintainability, tends to increase call chain lengths because it prohibits sending messages to results of other messages [73].

```
rcvrExp₁ max: argExp₁.
```

Assume a type-inference algorithm is analyzing this send in the context of a template T_1, and that the send may invoke our old friend, the `max:` method. The first time a use of `max:` is encountered, the strategy is simple: create a new template for `max:` and establish constraints between it and the above send in T_1. Later the type-inference algorithm may encounter a second use of `max:`

```
rcvrExp₂ max: argExp₂.
```

The second use may either be a syntactically different send in the target program or it may be the same send as the first use, but now being analyzed in the context of a different template T_2. Given these two uses of `max:` the crucial question is:

Can they share a `max:` *template or should they each have their own?*

Figure 24 illustrates this choice, which any type-inference algorithm must make when facing two send expressions invoking the same method. The sharing choice is crucial because it selects the type inferencer's trade-off between efficiency and precision. The stakes involved are all the higher because choosing implies a solid commitment: to preserve monotonicity, the type inferencer writes off the possibility of later changing a sharing decision from sharing to not sharing, or vice versa. Recall from the three-step description of the basic type-inference algorithm in Section 3.2.1 (page 37) that monotonicity is a fundamental part of constraint-based type inference: during analysis, the inferencer builds a monotonically growing network of type variables and constraints in which the types are growing monotonically. Monotonicity must be respected since (in general) the constraint-based framework offers no way to retract object types from type variables or remove constraints from the network, and since auxiliary mechanisms such as dynamic dispatch resolution (discussed in Section 4.1) rely on it. Consequently, once a type-inference algorithm has decided whether or not to use a shared template, the decision cannot later be changed, since it would involve removing constraints previously established, i.e., it would breach monotonicity.

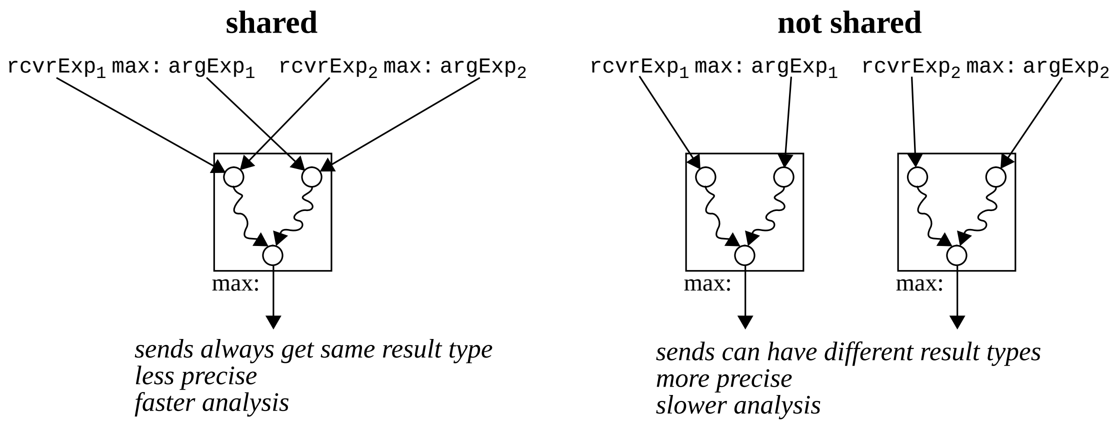


Figure 24. To share or not to share?

Given two sends to analyze, all the type-inference algorithms face the same question: should the sends share a template, or should they each have their own? Adaptive type-inference algorithms use type information to answer this question, thereby achieving better precision and efficiency than non-adaptive algorithms, which do not tap this source of information when answering the sharing question.

At first sight, changing from not sharing to sharing may seem possible without breaking monotonicity. However, the analysis of blocks, described in Section 4.4, complicates even this restricted change of sharing. To analyze blocks precisely, the type inferencer creates so-called closure object types, blocks that have been paired with templates. Leaving the closure object types that refer to the non-shared templates in the constraint network makes it impossible to eliminate any of the non-shared templates; retracting these closure object types from the network breaks monotonicity. Thus, while it may be possible in certain restricted cases to change sharing decisions, in general, sharing decisions are binding within the framework of constraint-based analysis.

In the light of the sharing question, we can summarize the previously described type-inference algorithms as follows:

- The basic algorithm always uses a shared template.
- The 1-level expansion algorithm uses a shared template if and only if the two sends are syntactically the same.
- The *p*-level expansion algorithm uses a shared template if and only if the two sends are the same after $p-1$ expansions of the program.

All these algorithms base the sharing decision entirely on static information such as the target program's syntax tree. In contrast, an *adaptive type-inference algorithm* will base the sharing decision on more than static information, using partial type information or whatever else it has available at the decision point. For a first example of an adaptive algorithm, consider one that employs a shared template if and only if:

$$\text{type}(\texttt{rcvrExp}_1, T_1) = \text{type}(\texttt{rcvrExp}_2, T_2) \;\wedge\; \text{type}(\texttt{argExp}_1, T_1) = \text{type}(\texttt{argExp}_2, T_2).$$

Hence, two sends invoking a method can share a template if and only if they supply the same receiver and argument types, respectively. This test precisely captures the circumstances in which a template can be shared without loss of precision caused by merging of different types. We call an algorithm that answers the sharing question based upon the above test *ideal*:

- It is precise because it never allows different types to merge.
- It is efficient because it avoids redundant analysis by always sharing a template whenever two or more sends invoke a method with the same actual arguments types.

Unfortunately, the algorithm is ideal in one more regard: it cannot be implemented. The reason is that the types it compares to decide whether to use a shared template are the very types being computed. At the time when the ideal algorithm must decide whether to use a shared template for two sends, it has incomplete knowledge of the sends' actual arguments' types: by monotonicity the types found in the network at that moment will be subsets of those found there when type inference eventually terminates. Comparing subsets of two actual types such as type($\texttt{rcvrExp}_1$) and type($\texttt{rcvrExp}_2$) provides no useful information: the subsets can be equal, yet the full types, known only at the end of type inference, may turn out to be unequal; the subsets can be unequal, yet the full types may turn out equal. Thus, the ideal algorithm cannot be implemented because it needs to know the full types before it has finished computing them. While the ideal algorithm cannot be implemented, there are several ways in which it can be approximated, so the situation is not hopeless.

The sharing decision in type inference resembles procedure cloning, a topic that Cooper, Hall, and Kennedy studied in the context of data-flow analysis of Fortran programs [34]. Cloning procedures improves precision of interprocedural data-flow analysis because each clone has fewer incoming calls and therefore more specific arguments. Cooper, Hall, and Kennedy give an algorithm for cloning procedures to expose more numeric constants and mention that their algorithm can be generalized to any forward data-flow problem.

In summary, adaptive algorithms have the advantage over non-adaptive algorithms that they can use (partial) type information to make better sharing decisions and therefore improve the trade-off between efficiency and precision. However, even adaptive algorithms must make their sharing decisions decisively to respect monotonicity. Sections 3.2.6, 3.2.7, and 3.2.8 study three specific adaptive algorithms.

3.2.6 The hash function algorithm

The hash function algorithm was the first adaptive type-inference algorithm developed for an object-oriented language [6]. It significantly improved precision *and* efficiency over the non-adaptive algorithms. As a result, precise types could be inferred efficiently for many Self programs for the first time. The algorithm takes its name from a hash function that plays a key role in controlling the behavior of the algorithm. The hash function maps a (send, template) pair to a hash value:

$$\text{hash: Send} \times \text{Template} \rightarrow \text{Hash\_value}.$$

The algorithm computes hash values to decide when templates can be shared: two send expressions (in the same or different templates) that invoke the same method *M* can share an *M*-template if and only if they have the same hash value. Since the hash function can depend on the types of the actual arguments in the (send, template) pairs, the resulting type-inference algorithm is adaptive.

The hash algorithm is actually a family of algorithms: one for each hash function. Different trade-offs between cost and precision of type inference can be obtained by applying different hash functions. However, not every function would work as a hash function. As explained in the previous section, at the time when sharing decisions must be made, the types in the constraint network are only partially known. They may later accumulate new members as type inference progresses. The hash function, in addition to guiding the partitioning of (send, template) pairs over templates for the invoked methods, must also “absorb” the uncertainty resulting from types that are known only partially. For example, a hash function that remains constant even if the involved types later grow satisfies this absorption requirement. (Less strict conditions could also be formulated, although we will refrain from doing so here.)

Before we describe a specific hash function that has been applied to Self programs, we generalize the hash function algorithm. Instead of using a hash function that computes a single hash value, we permit the hash function to compute a finite number of hash values for each (send, template) pair:

$$\text{hash: Send} \times \text{Template} \rightarrow \text{Hash\_value*}$$

(the asterisk denotes a sequence). The generalized hash function algorithm connects a given (send, template) pair to a template for each hash value computed by the hash function, possibly propagating only selected object types from the actual arguments to some of the templates. In this regard, the hash algorithm is different from the expansion algorithms: none of these would ever connect a (send, template) pair to more than one template for a given target method. The details will become clear when we describe a specific instance of the hash function algorithm below. Moreover, the cartesian product algorithm, described in Section 3.2.8, can be seen as an instance of the generalized hash function algorithm. (However, we prefer to initially describe the cartesian product algorithm independently of the hash function algorithm to emphasize the different perspectives behind the two algorithms.)

The remainder of this section describes a specific hash function that has been found to yield a reasonable compromise between precision and analysis cost for many programs. This hash function has been tested on Self programs (see [6]), but we believe it can be useful on other programs as well. Assume that a send expression *S* is being analyzed in the context of a template *T*, i.e., *T* is one of the templates for the method containing *S*. Assume furthermore that the type of the receiver of *S* is a set of *k* object types:

$$\text{type}(S.\text{receiver}, T) = \{\rho_1, \rho_2, \ldots, \rho_k\}.$$

The hash function computes a hash value for each object type in the receiver:

$$\text{hash}(S, T) = \{(\rho_1, S), (\rho_2, S), \ldots, (\rho_k, S)\}$$

For each method *M* that the send may invoke, the hash function algorithm creates *k* separate templates, one for each hash value (ρ_i, S). The algorithm connects the send to all *k* templates: when connecting to the *i*’th template, only the *i*’th object type in the receiver type, ρ_i, is propagated into the `self` type variable. Thus, each of the *k* templates is responsible for handling one possible receiver object type; see Figure 25.

To better understand this instance of the hash function algorithm, consider the significance of each component of a typical hash value (ρ_i, S).

- The first component, ρ_i, is an object type for a possible receiver. Thus, two send expressions invoking the same method can share a template only if they provide the same receiver object. This ensures that the type of `self` is always a singleton set (the effect is similar to that of customization [28]). Singleton types for `self` improve type-

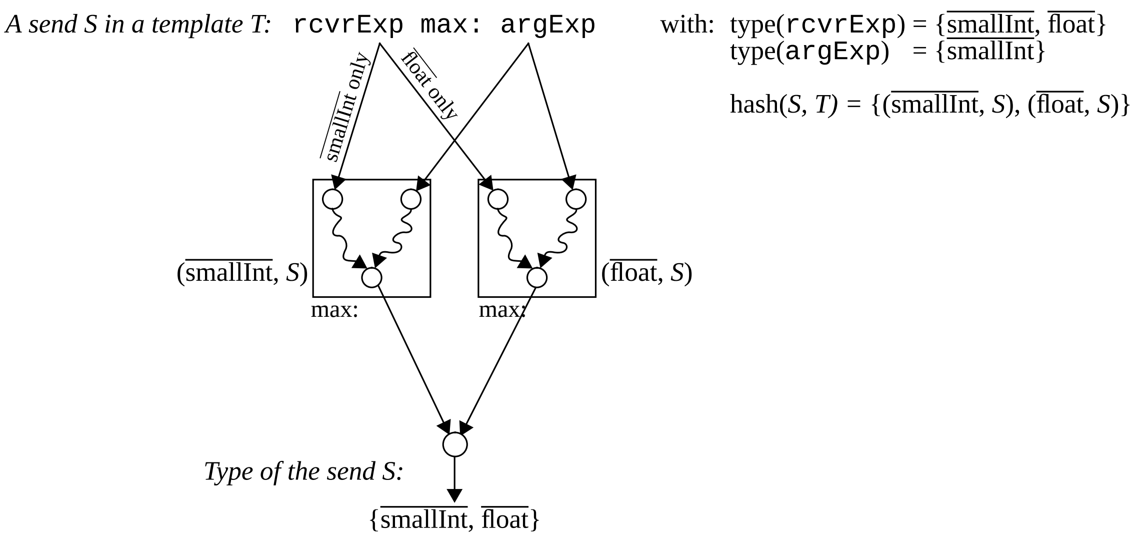

A send S in a template T: rcvrExp max: argExp
with: type(rcvrExp) = {smallInt, float}
type(argExp) = {smallInt}
hash(S, T) = {(smallInt, S), (float, S)}
smallInt only
float only
(smallInt, S)
(float, S)
max:
max:
Type of the send S:
{smallInt, float}

Figure 25. A specific instance of the hash function algorithm

The hash function applied here directs each possible receiver object type to a separate template for the invoked method. Furthermore, it prevents syntactically different send expressions from sharing a template.

inference precision in two ways. First, inherited methods are reanalyzed in the context of each object that inherits them. Second, sends to `self`, which tend to be common, can be analyzed more precisely because a single target can be found (except in cases with dynamic inheritance). In previous work, Palsberg and Schwartzbach obtained the same benefits by expanding away inheritance from the target program [86].

- The second component, *S*, is the (syntactic) send expression itself. Thus, different send expressions, which often supply different types of arguments, must connect to different templates. This component essentially simulates an implementation of the 1-level expansion. Hence, it also suffers from redundancy problems: even if two sends pass the same types of actual arguments, they must still be connected to different templates. One further refinement was used in [6]: when a send has no arguments, it is dropped from the hash values. This permits more sharing when it is obviously safe and therefore improves the efficiency of type inference.

One complication remains to be explained: the receiver type is generally not fully known during type inference, so how can it be used in the hash function? The answer is that although types are not fully known during inference, they are known to be monotonically growing sets. Hence, as long as the type-inference algorithm goes back and extends the analysis whenever a receiver type grows, everything works out fine: the hash algorithm simply connects the given send to additional templates. In Section 3.2.8, we will see that the cartesian product algorithm takes this idea even further.

3.2.6.1 Discussion of the hash function algorithm

The hash function does a good job of controlling polymorphism in the receiver. For example, the cascading `value:With:` methods in `defaultBehavior` (see Section 3.2.4.1) cause no problems when they fall through all the way to the bottom and return `self`. Consider these two sends:

```
444 value: nil With: nil With: nil.    "This send returns 444."
3.5 value: 100 With: 100 With: 100.    "This send returns 3.5."
```

The first send has `444` as the receiver. During type inference, the hash function forces the send to be connected to a `value:With:With:` template, *T*, for which type(`self`, *T*) = {$\overline{\text{smallInt}}$}. This pattern repeats for the `value:With:`, `value:`, and `value` sends until, finally, the type of the result is determined to be the type of `self` in the last template, i.e., $\{\overline{\text{smallInt}}\}$. Since the second send of `value:With:With:` has `3.5` as the receiver, it will yield a distinct set of templates in which the type of `self` is $\{\overline{\text{float}}\}$; thus, there will be no interference between the two sends, and their types can be precisely inferred.

While the hash function algorithm controls polymorphic receivers well, it does not excel on polymorphic arguments. In fact, the hash function does not even depend on argument types. Polymorphic arguments appear to be less common in Self than polymorphic receivers, but there are nevertheless important cases where the polymorphism is in the arguments and not the receiver, e.g., `ifTrue:False:` as we have previously seen. Another example of polymorphic arguments is found in methods that implement arithmetic operators such as “`+-*/`”. These operators are “doubly-dispatched,” a standard trick used to ensure that the types of both the receiver and argument are known in the method that implements the arithmetic operations [60]. Without a precise processing of polymorphic arguments, the hash function algorithm loses information when analyzing the double-dispatching code, since it is unable to keep distinct types separate when they appear as arguments.

To address these shortcomings, the algorithm described in [6] supplements the hash function with a small number of additional rules. The rules specify that templates should *never* be shared for a certain small group of methods, including the double-dispatching methods, `ifTrue:False:`, and a few others. The computational cost of never sharing these templates is affordable, since the methods are small and few. However, the rules affect the robustness of the type-inference algorithm, making it sensitive to how certain things in the Self world are coded.

3.2.7 The iterative algorithm

Plevyak and Chien [93, 94] developed an iterative adaptive type-inference algorithm and applied it to the Concurrent Aggregates language in the Illinois Concert System [30]. Concurrent Aggregates is a dynamically-typed, single-inheritance, object-oriented language. Their algorithm improved the precision of type inference for code with both parametric and data polymorphism; here, we consider the former aspect only (data polymorphism will be discussed in Section 4.3).

The best way to understand the iterative algorithm is by comparing it to the ideal algorithm. Recall from Section 3.2.5 that the ideal algorithm, upon encountering these two sends

$\texttt{rcvrExp}_1\ \texttt{max:}\ \texttt{argExp}_1.$ (in the context of a template T_1)
$\texttt{rcvrExp}_2\ \texttt{max:}\ \texttt{argExp}_2.$ (in the context of a template T_2)

permits use of a shared template if and only if

$$\text{type}(\texttt{rcvrExp}_1,T_1) = \text{type}(\texttt{rcvrExp}_2,T_2) \ \wedge\ \text{type}(\texttt{argExp}_1,T_1) = \text{type}(\texttt{argExp}_2,T_2).$$

The problem with the ideal algorithm is that it needs the types before they are computed. Plevyak and Chien found a way out of this problem by iterating type inference. The 0’th iteration is simply the basic algorithm. In subsequent iterations, the ideal algorithm’s sharing test is approximated by using the types of the previous iteration. That is, the two sends can share a template in iteration p if and only if:

$$\text{type}_{p-1}(\texttt{rcvrExp}_1,T_1) = \text{type}_{p-1}(\texttt{rcvrExp}_2,T_2) \ \wedge\ \text{type}_{p-1}(\texttt{argExp}_1,T_1) = \text{type}_{p-1}(\texttt{argExp}_2,T_2)\ .$$

In other words, during the p’th iteration, two sends share a template if and only if the sharing does not result in any loss of precision according to the types inferred in iteration $p-1$.

In their earlier paper [93], Plevyak and Chien described their algorithm using the concepts of “entry sets” and “splitting”; in the later paper [94], the term “contour” from [107] is used instead of “entry set.” It is helpful to know, when reading these papers, that entry sets correspond to templates and splitting corresponds to the creation of additional templates for a given method.

3.2.7.1 Discussion of the iterative algorithm

The iterative algorithm is precise since it can handle arbitrarily deep polymorphic call chains, given enough iterations: after p iterations, the precision can be as good as that of the p-level expansion algorithm. In particular, the iterative algorithm can infer precise types for the cascading `value:With:With:` sends, double-dispatching methods, conditional statements, and the `max:` method.

The iterative algorithm achieves a given precision at a lower expected cost than a similarly precise expansion algorithm. The reason is that the iterative algorithm "selectively" expands the target program only as is needed to avoid the merging of non-equal types at calls, whereas the non-adaptive expansion algorithms apply the same level of expansions uniformly throughout the target program.

Iteration, while improving precision and being more efficient than a correspondingly precise expansion algorithm, also has drawbacks:

- *Initial hurdle*. The iterative algorithm improves precision from iteration to iteration. The first iteration is the least precise, being equivalent to the basic algorithm. In practice, for many Self programs the basic algorithm's precision is insufficient to ensure termination before running out of memory (see Section 3.4). Consequently, the iterative algorithm in its pure form suffers from the same problem, and may never get through the first iteration. To avoid this problem, it is, of course, possible to use the 1-level expansion algorithm or any other sufficiently precise algorithm for the first iteration.

- *Convergence*. It can be hard to know when to stop iterating. In principle, once the number of iterations is greater than or equal to the length of the longest polymorphic call chain, convergence should happen, i.e., the inferred types should no longer change. In reality, block specialization, data polymorphism, and recursion may interact to make the situation more complex; see [93] and Chapter 5.

- *Overhead*. Each iteration performs a complete global analysis of the target program. Thus, compared with a precise single-pass approach such as the cartesian product algorithm (described next), the iterative algorithm may be slower by a factor roughly equal to the number of iterations required.

- *Implementation complexity.* Mapping type information across iterations is non-trivial, because the program is expanded differently.

Section 4.1.3 returns to iteration in a more general setting. It revisits the above problems both in general and for several specific analyses, and contrasts iteration with an alternative we have termed "integration."

We are now ready to describe the cartesian product algorithm, the main contribution of this chapter. It is an example of what we call an integrated algorithm in Section 4.1.3.

3.2.8 The cartesian product algorithm

The expansion algorithms and the iterative algorithm try to obtain precision and efficiency by partitioning sends according to the types of their actual arguments. The iterative algorithm reaches the goal, but incurs the overhead of iteration to timely harvest the type information that it needs during the analysis. The cartesian product algorithm (CPA for short) differs fundamentally. It does not partition sends, but instead turns the analysis of *each* send into a case analysis. To analyze a send, CPA computes the cartesian product of the types of the actual arguments. Each tuple in the cartesian product is analyzed as an independent case. This case analysis makes exact type information immediately available for each case, thus eliminating the need for iteration. In turn, the type information is used to ensure both precision (by avoiding type merges) and efficiency (by sharing cases to avoid redundant analysis). In the following description of CPA we compare it mainly against the iterative algorithm, since these two are the most powerful algorithms.

The idea behind CPA is best understood by going back to the analogy between program execution and program analysis. During program execution, activation records are always created "monomorphically," simply because each slot contains a single object. Consider, for example, a polymorphic send expression that invokes the `max:` method with integer or float receivers. This means that sometimes the send invokes `max:` with an integer receiver, and other times it invokes it with a float receiver. But in any particular invocation the receiver is *either* an integer or a float: it cannot be both. We summarize this observation as follows:

There is no such thing as a polymorphic message, only polymorphic send expressions.

CPA, unlike the other algorithms, exploits this observation. All formal arguments in the templates that CPA creates have monomorphic types. Given a send expression such as

```
    rcvrExp max: argExp.
```

Let R and A denote the types of the receiver and arguments, respectively:

$$R = \text{type}(\texttt{rcvrExp}) \text{ and}$$
$$A = \text{type}(\texttt{argExp}).$$

Suppose it is (somehow) known that:

$$R = \{\rho_1, \rho_2, \ldots, \rho_s\} \text{ and}$$
$$A = \{\alpha_1, \alpha_2, \ldots, \alpha_t\} .$$

To analyze this send, CPA computes the cartesian product of the receiver type and all argument types[†]. In the present case there is only one argument, so the cartesian product is a set of pairs; in general it is a set of (k+1)-tuples, where k is the number of (non-receiver) arguments.

$$R \times A = \{ (\rho_1, \alpha_1), \ldots, (\rho_1, \alpha_t), \ldots, (\rho_i, \alpha_j), \ldots, (\rho_s, \alpha_1), \ldots, (\rho_s, \alpha_t) \} .$$

Next, CPA propagates each $(\rho_i, \alpha_j) \in R \times A$ into a separate `max:` template. If a `max:` template already exists for a given (ρ_i, α_j) pair, it is reused; if no such template exists, a new one is created and made available for this and future (ρ_i, α_j) pairs in other sends. Finally, the type of the send is obtained as the union of the result types of the templates that the send was connected to. Figure 26 illustrates the situation with concrete object types for a single send.

In the Self implementation, each method has a repository of templates. All templates for a method *M*, no matter which send invoking *M* caused their creation, are stored in *M*'s repository. The repositories are indexed by (k+1)-tuples to allow fast retrieval of the template corresponding to a given tuple. Using these structures, the analysis of a send proceeds as follows:

1. Determine the target methods that the send may invoke.
2. Generate the cartesian product (lazily, as described below in Section 3.2.8.2).
3. For each target method *M* and each tuple $(\alpha_0, \ldots, \alpha_k)$ in the cartesian product, look up $(\alpha_0, \ldots, \alpha_k)$ in *M*'s template repository. If no matching template is found, create one and add it to the repository.
4. Connect the send to the—new or old—template.

To obtain an efficient algorithm, it is necessary to maintain per-method repositories of templates rather than per-send repositories, to ensure that different sends whose cartesian products have tuples in common can share templates. It is worth emphasizing that the template repositories are not filled up "in advance." Rather, templates are added to them gradually, as new argument combinations (tuples) are encountered during analysis of all the sends in the target program. In fact, it is not possible to know in advance which templates are going to be needed. It only becomes clear gradually, as the analysis progresses.

3.2.8.1 Assessing the cartesian product algorithm

The cartesian product algorithm is *precise* in the sense that it can analyze arbitrarily deep polymorphic call chains without loss of precision. CPA avoids precision loss because it creates monomorphic templates and never allows

[†] In general, of course, these types will not be fully known during type inference; this lack of information was the motivation behind the iterative algorithm. In Section 3.2.8.2, we explain how the cartesian product algorithm can nevertheless use the types without iterating or even approximating.

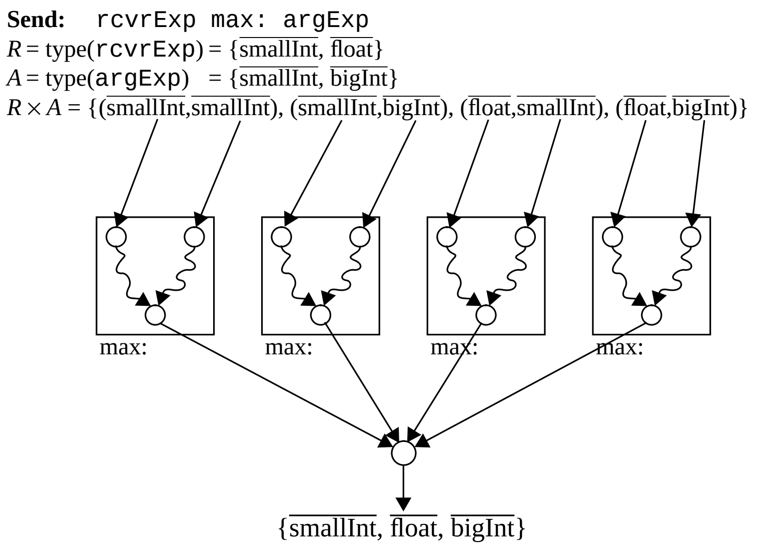

Send: rcvrExp max: argExp
R = type(rcvrExp) = {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}
A = type(argExp) = {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}
$R \times A$ = {($\overline{\text{smallInt}}$,$\overline{\text{smallInt}}$), ($\overline{\text{smallInt}}$,$\overline{\text{bigInt}}$), ($\overline{\text{float}}$,$\overline{\text{smallInt}}$), ($\overline{\text{float}}$,$\overline{\text{bigInt}}$)}
max:
max:
max:
max:
{$\overline{\text{smallInt}}$, $\overline{\text{float}}$, $\overline{\text{bigInt}}$}

Figure 26. The cartesian product algorithm

The cartesian product algorithm forms the cartesian product of the receiver type and argument types of the send it is analyzing. Each member of the product is propagated to a template reserved exclusively for that receiver and argument combination. The result type of the send is obtained by collecting the contributions from each template.

different tuples from two or more sends to be propagated into the same template. In other words, there is no merging, no matter how deep the call chain is.

The cartesian product algorithm is *efficient* because it avoids redundant analysis. As we have seen before, the key to avoiding redundancy is sharing. Each time a (ρ_i, α_j) pair can occur in some call of `max:` it is connected to the same shared template. Hence, each combination of a specific receiver and argument is analyzed only once; see Figure 27. In contrast, the iterative algorithm may generate templates with overlapping types such as:

Template 1:	type(`self`)	= {$\overline{\text{smallInt}}$, $\overline{\text{smallInt}}$}
	type(`argExp`)	= {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$, $\overline{\text{rational}}$}
Template 2:	type(`self`)	= {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}
	type(`argExp`)	= {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}

In the above situation, CPA generates more templates than the iterative algorithm (six versus two). This does not necessarily mean that CPA is slower, since the templates it generates have singleton types for the receiver and arguments, and hence can be processed faster than templates with larger types.

Merging the result types of the templates that CPA connects a send to (see Figure 27) causes a loss of information. This loss, however, is “built into” the expansion algorithms and the iterative algorithm since these algorithms never separate the cases in the first place, i.e., it is not an additional loss incurred by the cartesian product algorithm. One could consider avoiding this loss by eliminating the merge. However, the consequence would be an increase in analysis cost, since there would be more cases to deal with “downstream.”

3.2.8.2 How the cartesian product algorithm can use types without iterating

We have yet to explain how the cartesian product algorithm can use the types of arguments and receivers while still in the process of computing them. The iterative algorithm found a way out of this dilemma by iterating and approximating the types with those computed in the previous iteration. CPA avoids the dilemma altogether, using neither approximations nor iteration.

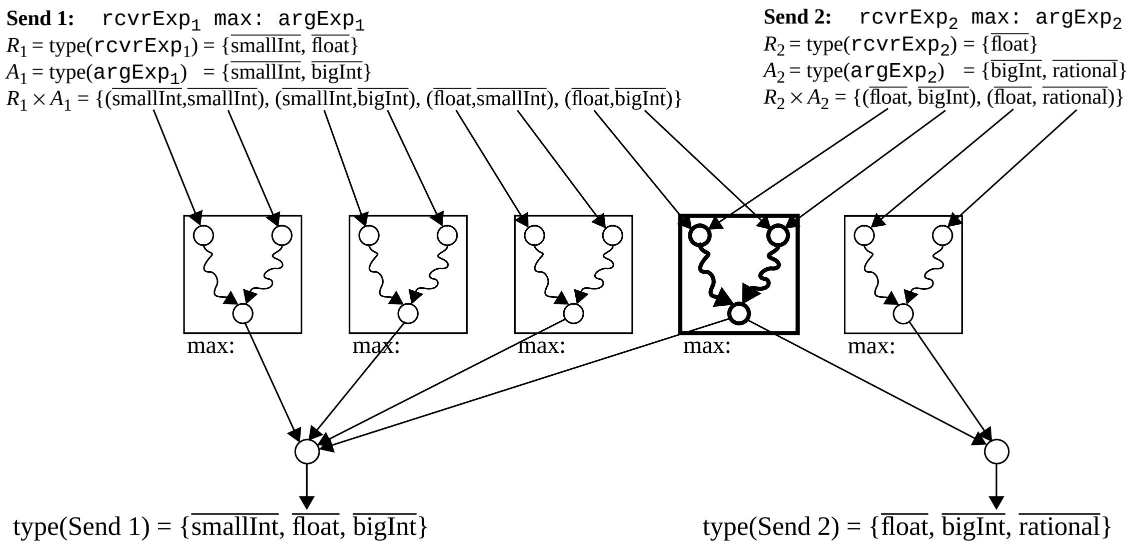

Send 1: rcvrExp1 max: argExp1
R1 = type(rcvrExp1) = {smallInt, float}
A1 = type(argExp1) = {smallInt, bigInt}
R1 × A1 = {(smallInt,smallInt), (smallInt,bigInt), (float,smallInt), (float,bigInt)}
Send 2: rcvrExp2 max: argExp2
R2 = type(rcvrExp2) = {float}
A2 = type(argExp2) = {bigInt, rational}
R2 × A2 = {(float, bigInt), (float, rational)}
max:
max:
max:
max:
max:
type(Send 1) = {smallInt, float, bigInt}
type(Send 2) = {float, bigInt, rational}

Figure 27. How the cartesian product algorithm achieves efficiency

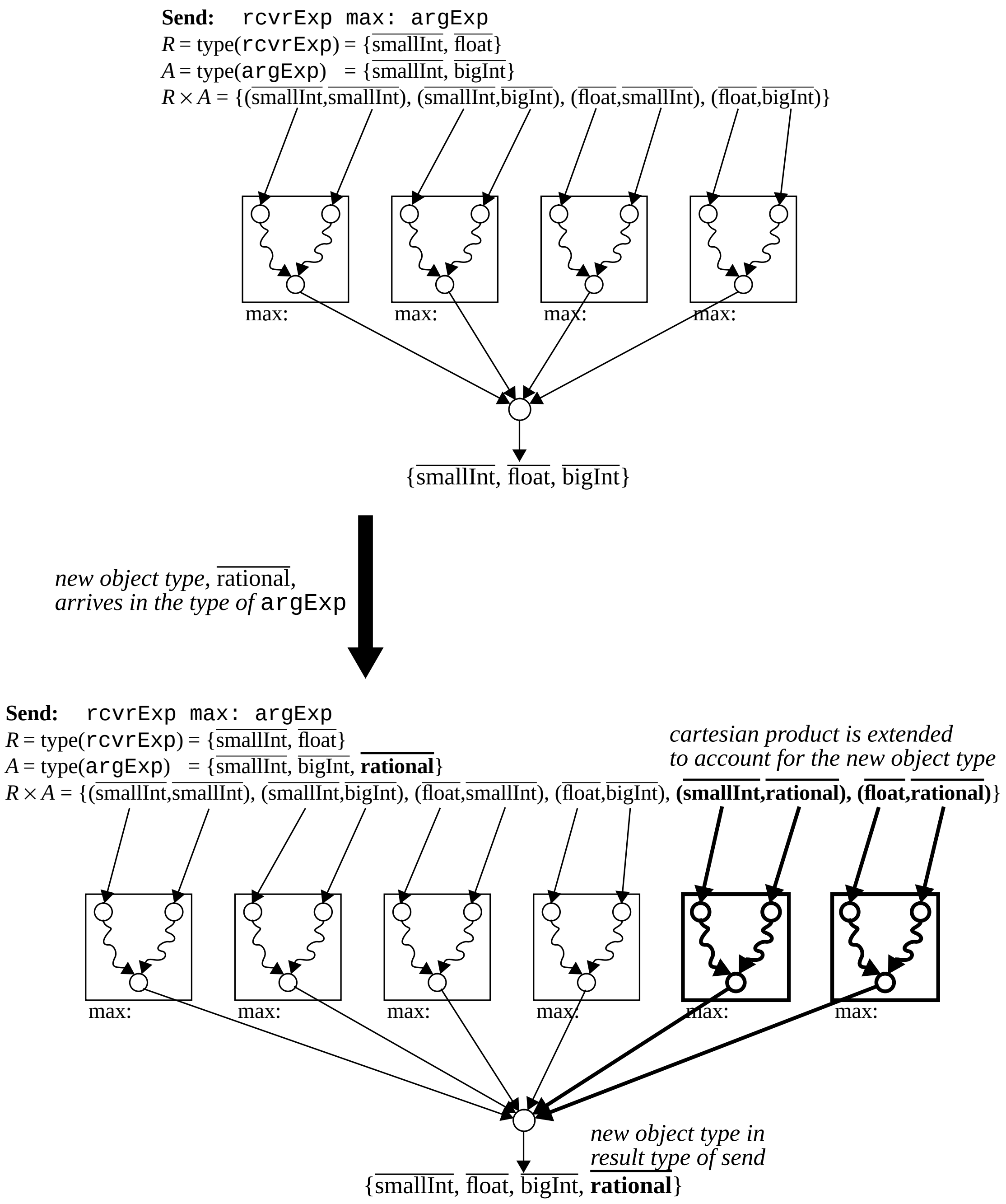
CPA achieves efficiency by sharing templates for equal tuples across all sends. Above, the `max:` template for ($\overline{\text{float}}$, $\overline{\text{bigInt}}$) is shared between the two sends. Since sharing happens only for equal tuples, it causes no precision loss. For example, the $\overline{\text{rational}}$ object type found in the second send's argument type cannot pollute the result type of the first send — despite the shared template.

How is this achieved? The key is that, while the iterative algorithm compares types to find out whether or not to share templates, CPA computes cartesian products instead. The cartesian product, unlike a type comparison, can be computed correctly, albeit gradually, even if the types are known only partially until the very end of the analysis. Specifically, when a send is first processed, the cartesian product of the *current* members of the receiver and argument types is computed and connections to relevant templates are made. If one or more of the types later grow, CPA returns to the send and extends the cartesian product with the new combinations that are now possible. This approach yields a correct solution since types grow monotonically during type inference.

For illustration, assume that CPA has built the constraint network that was shown in Figure 26, but a new object type, $\overline{\text{rational}}$, has just arrived in the type of the argument; see Figure 28. The algorithm first extends the cartesian product with two new pairs: ($\overline{\text{smallInt}}$, $\overline{\text{rational}}$) and ($\overline{\text{float}}$, $\overline{\text{rational}}$). Then it propagates these pairs into (new or old) `max:` templates and collects the new possible result types.

Two key properties allow CPA to avoid iteration. First, the cartesian product is a *monotone* function. If either R or A grows, the cartesian product $R \times A$ grows. Second, the cartesian product is applied in a "monotone context": if a cartesian product grows, the correct compensating action is to grow the constraint network. Because CPA manages to preserve monotonicity, it can run in a single pass, yet will on no occasion "regret" a previous action. The iterative algorithm, in contrast, breaks monotonicity because comparing types for equality is not a monotone function. To buy back monotonicity (within each iteration), it pays the price of iteration. The expansion algorithms, being non-adaptive, never endanger monotonicity since their sharing decisions do not involve type information. Finally, the hash algorithm with the hash function described in Section 3.2.6 avoids conflicts with monotonicity for similar reasons as does CPA: growing types introduce new cases, but the test it applies when deciding whether to share templates remains unaffected by growing types.

We have completed our description of the cartesian product algorithm. Section 3.3 discusses it in a larger context, and Section 3.4 gives empirical data for it and the previous expansion algorithms. In closing, it should be mentioned that in work done recently at Aarhus University following the present author's work, the cartesian product algorithm has also been implemented for the Beta language. Details can be found in [50].



Send: rcvrExp max: argExp
R = type(rcvrExp) = {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}
A = type(argExp) = {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}
$R \times A$ = {($\overline{\text{smallInt}}$,$\overline{\text{smallInt}}$), ($\overline{\text{smallInt}}$,$\overline{\text{bigInt}}$), ($\overline{\text{float}}$,$\overline{\text{smallInt}}$), ($\overline{\text{float}}$,$\overline{\text{bigInt}}$)}
max:
max:
max:
max:
{$\overline{\text{smallInt}}$, $\overline{\text{float}}$, $\overline{\text{bigInt}}$}
new object type, $\overline{\text{rational}}$, arrives in the type of argExp
Send: rcvrExp max: argExp
R = type(rcvrExp) = {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}
A = type(argExp) = {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$, $\overline{\textbf{rational}}$}
$R \times A$ = {($\overline{\text{smallInt}}$,$\overline{\text{smallInt}}$), ($\overline{\text{smallInt}}$,$\overline{\text{bigInt}}$), ($\overline{\text{float}}$,$\overline{\text{smallInt}}$), ($\overline{\text{float}}$,$\overline{\text{bigInt}}$), ($\overline{\textbf{smallInt}}$,$\overline{\textbf{rational}}$), ($\overline{\textbf{float}}$,$\overline{\textbf{rational}}$)}
cartesian product is extended to account for the new object type
max:
max:
max:
max:
max:
max:
new object type in result type of send
{$\overline{\text{smallInt}}$, $\overline{\text{float}}$, $\overline{\text{bigInt}}$, $\overline{\textbf{rational}}$}

Figure 28. How the cartesian product algorithm tolerates growing types

The cartesian product algorithm tolerates “growing” types in this way: when the type of a receiver or an argument grows, the cartesian product is extended, and the send is connected to additional templates. The new structures are shown in bold above.

3.3 Discussion

The cartesian product algorithm, the iterative algorithm, and the hash function algorithm are all examples of adaptive type-inference algorithms. These algorithms are related in interesting ways and can be combined to obtain still better adaptive algorithms. Section 3.3.1 discusses some of the possibilities. Following that, Section 3.3.2 looks at scaling issues and describes an extension to the cartesian product algorithm to improve performance when analyzing extremely polymorphic sends. Finally, Section 3.3.3 demonstrates by means of an example how CPA sometimes achieves better precision than any of the other algorithms described in this chapter.

3.3.1 A spectrum of type-inference algorithms

The specific instance of the hash function algorithm presented in Section 3.2.6 is a hybrid between the 1-level expansion and the cartesian product algorithm. It applies case analysis on receiver types like CPA, but uses the 1-level expansion to control parametric polymorphism in the non-receiver arguments. Like the two algorithms that this instance of the hash function algorithm inherits from, it is monotonic and can be implemented in one pass.

To illustrate the range of possible type-inference algorithms, imagine a hybrid between the iterative and the cartesian product algorithm. This hybrid algorithm may generate the cartesian product for some arguments of some sends, and apply iteration to the remaining arguments. The resulting algorithm, of course, cannot be implemented in one pass since its iterative aspects do not preserve monotonicity.

The cartesian product algorithm, the ideal algorithm, and the iterative algorithm can all be viewed as instances of the hash function algorithm: the hash function for CPA is the cartesian product, the hash function for the ideal algorithm is the identity function, and the hash function for the iterative algorithm returns the types computed in the previous iteration. In this perspective, the cartesian product algorithm and the iterative algorithm (or any expansion algorithm) represent opposite extremes in the following sense. When analyzing a single send in the context of some template, a sound inference algorithm must analyze all possible combinations of actual arguments, i.e., must cover the cartesian product of the actual argument types[†]. The two extreme approaches to ensuring coverage are:

- *Maximal break-down.* CPA achieves coverage by propagating each tuple in the cartesian product into a separate template. Thus, CPA performs a case analysis with a maximal number of cases, each of which is minimally complex.
- *Minimal break-down.* The iterative algorithm achieves coverage by propagating the send's full actual argument types into a single template. Thus, the iterative algorithm performs no case analysis (or a degenerate one consisting of a single case).

In between these extremes, a spectrum of possibilities exist. Some of these algorithms can be implemented in one pass (e.g., the original hash function algorithm), whereas others cannot (e.g., CPA/iterative hybrid). Furthermore, the choice of break-down level can be made on a per-send basis; in the next section, we describe an extension to the cartesian product algorithm that exploits this idea to analyze extremely polymorphic sends more efficiently. By outlining this spectrum of possible algorithms, we hope that future developers of type-inference systems will find guidance in designing an algorithm that fits their particular needs.

3.3.2 Scaling issues

Both the iterative and cartesian product algorithms may have problems analyzing large programs that are extremely rich in polymorphism. We give two scenarios. The first favors CPA. For simplicity, consider a method M with a single argument (it is straightforward to generalize to an arbitrary number of arguments). Assume that throughout the program, the method is called with receivers from the set $P = \{\rho_1, \rho_2, \ldots, \rho_s\}$ and with arguments from the set $Q = \{\alpha_1, \alpha_2, \ldots, \alpha_t\}$. In the worst case, the iterative algorithm may generate an exponential number of M-templates, one for each subset of P and Q, i.e., 2^{s+t} templates (there are 2^s subsets of P and 2^t subsets of Q). CPA, in contrast, gener-

[†] Sometimes the structure of a send permits elimination of certain tuples from the cartesian product of the actual arguments. For example, if the type of `x` is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$ and the send is `x+x`, only the tuples $(\overline{\text{smallInt}}, \overline{\text{smallInt}})$ and $(\overline{\text{float}}, \overline{\text{float}})$ are relevant; the tuples $(\overline{\text{smallInt}}, \overline{\text{float}})$ and $(\overline{\text{float}}, \overline{\text{smallInt}})$ represent cases that cannot occur at run time, thus can be discarded during analysis to improve precision. In general, we say that the arguments are *coupled*. Coupling is not restricted to identical objects, but applies whenever objects are (somehow) bound to have the same object type. We have implemented a conservative test for coupling, but found it to have only a minor effect on typical Self programs. However, this observation may not generalize to other languages.

ates at most a polynomial number of templates, one for each member of $P \times Q$, i.e., st templates. So far, though, we have not encountered this worst-case scenario in the programs we have analyzed.

The second scenario favors the iterative algorithm. Assume the target program contains a send whose receiver and 18 arguments (!) have the type $\{\overline{\text{smallInt}}, \overline{\text{bigInt}}, \overline{\text{float}}\}$. The cartesian product then has size $3^{19} \approx 10^{9}$. In this extreme case, even attempting to compute the cartesian product, not to mention creating 10^{9} templates, is a bad idea. Again, we have not encountered such extreme polymorphism, but we do apply the following test to determine if we are in danger of starting an unreasonably large computation:

- For each receiver or actual argument type *T*, test if size(*T*) ≤ *c*, where *c* is some (small) number.

Useful values of *c* seem to be in the range 3 to 5, perhaps using different values for receivers and arguments, and possibly letting the value depend on the number of arguments of the send (if there are more arguments, a smaller limit is appropriate). We say that an actual argument or receiver of a send that fails the test is *megamorphic*. When megamorphic arguments occur, the full cartesian product should no longer be computed. Instead, the type inferencer creates tuples of the form (α_0, α_1, *, α_3) where a star indicates a megamorphic argument. When the inferencer propagates such a tuple into a template, it expands the star to the full megamorphic type of the particular actual argument.

Megamorphism is a property of actual arguments of sends in templates. Thus, a syntactic send may have an argument that evaluates to many different kinds of objects, yet during type inference the argument remains non-megamorphic because its object types are distributed over different templates of the method containing the send. For example, the one-armed conditional method

```
ifTrue: b = ( ifTrue: b False: nil )
```

in `traits boolean` may be invoked with dozens of different blocks throughout a target program. Still, the `b` argument of the `ifTrue:False:` send expression usually remains non-megamorphic because the type inferencer creates many `ifTrue:` templates, in each of which the type of b is quite small (usually even monomorphic). Of course, the union over all `ifTrue:` templates of the types of `b` will be a type with dozens of different blocks.

The contraction of megamorphic arguments is an effective technique to avoid large cartesian products, e.g., the 10^9 cases mentioned above can be collapsed to a single monovariant analysis. The cost, of course, may be precision loss. To limit the loss, templates should not be shared between sends with and without megamorphic arguments to avoid “polluting” the non-megamorphic cases. The precision loss can be further limited by applying expansions or even iteration to the (hopefully few) megamorphic cases.

In practice, the exact choice of how one deals with megamorphism does not seem to matter much. We offer two possible explanations. First, megamorphism is rare. Measuring on a number of Self programs, we found that 0-4% of the sends have a megamorphic argument if the limit is $c = 3$. Thus, even if some precision is lost at these sends, the overall consequences are limited. Second, it seems plausible that a programmer who writes a send with two or three kinds of arguments may have distinct cases in mind, but if there are a dozen or more kinds, the programmer is likely exploiting some uniform behavior that in turn limits precision loss when the cases are merged during analysis.

In the implemented type inferencer, we also use the *-contraction to avoid customizing on the (formal) arguments that the invoked method ignores: an ignored argument cannot affect precision of type inference, so nothing is gained from customizing on it. This optimization may be more important in Self than in other languages, since many iterator methods (such as `do:` for lists) pass two arguments, while only one is used by the iterated block.

3.3.3 A case where the cartesian product algorithm improves precision

Even though both the iterative and the cartesian product algorithm avoid precision loss caused by merging types, CPA’s monomorphic analysis of each argument combination sometimes gives it a precision edge. Consider this method:

```
mod: arg = ( self-(arg*(self div: arg)) ).
```

Assume that `div:` denotes integer division, i.e., a division operator that *fails* if the receiver or argument is not an integer. Consider the analysis of a send, `x mod: y`, in a context where

$$\text{type}(\texttt{x}) = \text{type}(\texttt{y}) = \{\overline{\text{smallInt}}, \overline{\text{float}}\}.$$

The iterative algorithm will infer that the type of `x mod: y` is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$ whereas the cartesian product algorithm will infer the more precise type $\{\overline{\text{smallInt}}\}$. Here's how:

- The iterative algorithm analyzes `x mod: y` by connecting it to a `mod:` template in which the type of both `self` and `arg` is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$. Then it proceeds to find that the type of `self div: arg` is $\{\overline{\text{smallInt}}\}$ (since `div:` fails on floats). So far, the precision is optimal. Next, the iterative algorithm determines the type of `arg*(self div: arg)`. Since `arg` has type $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$, the type of the product is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$. In turn, the type inferred for `self-(arg*(self div: arg))` is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$, and finally the iterative algorithm finds that the type of `x mod: y` is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$.

- CPA analyzes `x mod: y` by connecting it to four templates, one for each tuple in the cartesian product of the types of `x` and `y`. The template in which both `self` and `arg` have type $\{\overline{\text{smallInt}}\}$ straightforwardly has result type $\{\overline{\text{smallInt}}\}$. The other three templates have empty result types, since in each of them the type of `self div: arg` is empty (because `div:` fails on floats and there is no integer "backup"). Hence, CPA combines $\{\overline{\text{smallInt}}\}$ and three empty result types, to find that the type of `x mod: y` is $\{\overline{\text{smallInt}}\}$.

The `mod:` method was chosen to expose a property of CPA, but similar structures can conceivably be found embedded in more complex situations. It can be argued that enhancements of the iterative algorithm allow such cases to be analyzed precisely. For example, Graver and Johnson use a case analysis within methods to analyze similar structures precisely [48]. We find it preferable, though, to strive for the best precision available using our first tool, the constraints, and consider other enhancements only as they are truly needed.

3.4 Empirical results

We have implemented and tested six type-inference algorithms: the basic, the *p*-level expansion (for *p* = 1, 2, 3, and 4), and the cartesian product algorithm. Previously, our system also implemented the specific instance of the hash function algorithm described in Section 3.2.6. However, support for this algorithm was phased out after we implemented the more powerful cartesian product algorithm, so the following empirical data exclude the hash function algorithm. We also have no empirical data on the iterative algorithm, since it has not been implemented for Self. We considered comparing our measurements for the other algorithms against the numbers published by Plevyak and Chien [93] for the iterative algorithm. In the end, we decided against it, since such a comparison would not be very meaningful: Plevyak and Chien implemented their algorithm in a different language, analyzed a different language, and chose different benchmarks.

Our implementation of the expansion algorithms and the cartesian product algorithm consists of 15,000 lines of Self source code that, in addition to type inference, performs tasks such as grouping of objects, SSA transformation, incremental analysis, and application extraction. We applied the six inference algorithms to the Self programs shown in Table 7. These programs were written by different programmers, and although some of them were written with compiler benchmarking in mind (most notably Richards), none of them were written to accommodate type inference. Some of the benchmarks, in fact, predate the type inferencer.

The programs were analyzed unmodified, with one exception: for the least powerful type-inference algorithms we had to patch the `_Mirror` primitive to fail instead of returning a mirror (mirrors are a way to write reflective code in Self and reflection is only partially supported by the type inference implementation). Without the patch, both the basic and the 1-level expansion algorithm would get lost in the reflective part of the Self system, continuing to build ever-larger constraint graphs until memory was exhausted. Even with the patch, the basic algorithm could not infer types for the PrimMaker benchmark. Thus, the tables and graphs below contain no data for the basic algorithm applied to PrimMaker. The mirror patch was removed when running the other inference algorithms, since they are precise enough to avoid falling into the reflective trap†.

† The pure form of the iterative algorithm would also need the patch, since its first iteration is equivalent to the basic algorithm. Moreover, the execution times later reported for the basic algorithm will, for the same reason, be lower bounds for the iterative algorithm.

Benchmark	Appl. size[a]	Total size[b]	File size[c]	Description
HelloWorld	1	10	107	Actually "Hello Self World!" According to Kernighan and Ritchie, this program is the first to write in any language [69]
Factorial	1	1,044	2,069	Recursive definition of the factorial function; mainly tests integer arithmetic
Richards	400	1,285	2,482	Operating system simulator; originally a BCPL program by Martin Richards [58]
DeltaBlue	500	1,359	2,526	DeltaBlue multiway constraint solver algorithm developed at the University of Washington [103]
Diff	300	2,064	3,659	Self implementation by Mario Wolczko of the UNIX® file comparison utility `diff` [57]
PrimMaker[d]	1,100	2,242	4,239	Generator of Self and C stubs from textual descriptions of C functions [58]

Table 7. Benchmark programs

a. Approximate lines of code in method bodies (i.e., expressions), excluding "standard code" such as integers, lists, etc. The line counts exclude blank lines.

b. Number of lines in the bodies of the methods that our application extractor (see Chapter 6) deems are part of the application; includes methods in standard objects such as integers, lists, and booleans.

c. Number of lines in the source file produced when the application extractor is applied to the application; these counts, unlike "Appl. size" and "Total size," include blank lines and definitions of instance variable, parents slots, etc.

d. We use the January 1994 version of PrimMaker—later versions manipulate *annotations* [118], a reflective concept that our type inference implementation does not support.

Like most programs, the ones in Table 7 are not self-contained. They reuse and extend existing objects and data structures. For example, the `factorial` method is only one line of source, but it needs the bigInt implementation, which comprises hundreds of lines. To obtain a more useful measure of program size, we applied the extractor described in Chapter 6, and measured the size of the resulting self-contained source files. The column labelled "Total size" in Table 7 gives the sizes of the extracted applications when counting method bodies only (i.e., statements and expressions). The column labelled "File size" gives the sizes when counting everything in the extracted files, including blank lines, definitions of instance variables, formal arguments, and parent slots.

The extractor uses type information and has the property that less precise type information forces it to extract more. We plugged each of the six type-inference algorithms into the extractor and measured how many lines and methods were extracted for each test program. Table 8 shows the number of lines in the extracted files and Table 9 the number of methods[†]. In addition, Figure 29 shows a plot of the data from Table 9. This plot, and all subsequent ones, omits the atypical and trivial HelloWorld benchmark. To account for the different sizes of the benchmarks, in the plot each benchmark is normalized by the number of methods extracted when using CPA. In general, CPA delivers the smallest extractions, indicating that it is the most precise inference method. On Diff, the expansion algorithm improves significantly when p is increased from 1 to 2, and shows little additional gain past 2. The *increasing* line counts in Table 8 from $p = 1$ to $p = 2$ on Factorial, Richards, DeltaBlue, and PrimMaker are explained by the removal of the reflective patch. The numbers in Table 8 and 9 are important beyond indicating type-inference precision. A compiler based on type inference compiles fewer methods if it uses CPA than any of the other algorithms (note: the current Self compiler does not use type inference; see Section 7.3).

Table 10 and Figure 30 compares CPU times of the type-inference algorithms, measured on a 167 MHz UltraSPARC. To limit the variability caused by Self's dynamic compilation and reoptimization, we inferred types for each benchmark ten times, reporting the average CPU time over the last three repetitions. To focus on the type-inference algorithms per se, object grouping and lookups were cached from execution to execution. The numbers in the table show

[†] In a previous publication [3], we reported similar measurements for some of the benchmarks used here. The previously published numbers differ somewhat from those given here; some of the benchmarks have changed (although they still have the same names), and the type inferencer's recursion handling was different in the previous study; see Section 5.3.4.

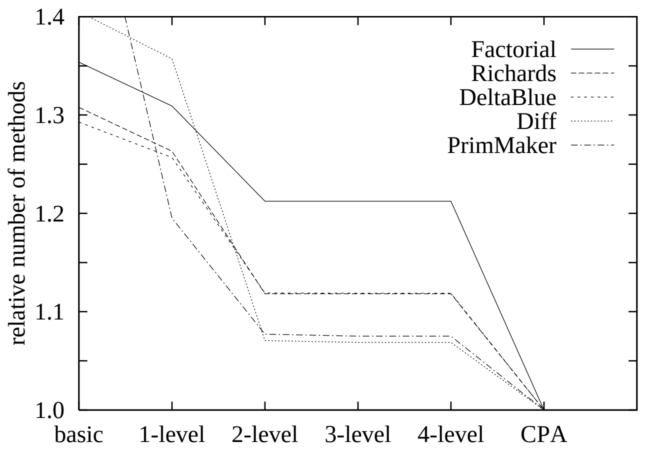


Figure 29. Number of methods extracted (normalized to CPA)

	basic/0-level	1-level	2-level	3-level	4-level	CPA
HelloWorld	107	107	107	107	107	107
Factorial	1,846	1,835	2,239	2,239	2,239	2,069
Richards	2,250	2,233	2,652	2,652	2,652	2,482
DeltaBlue	2,273	2,262	2,709	2,707	2,707	2,526
Diff	5,748	3,808	3,746	3,744	3,744	3,658
PrimMaker	not available[a]	4,074	4,327	4,325	4,325	4,239

Table 8. File size of extracted applications (number of lines)

a. Even with the mirror patch in place, the basic algorithm runs out of memory before finishing the analysis of PrimMaker.

	basic/0-level	1-level	2-level	3-level	4-level	CPA
HelloWorld	7	7	7	7	7	7
Factorial	727	703	651	651	651	537
Richards	820	792	741	741	741	627
DeltaBlue	861	837	793	791	791	666
Diff	1,412	1,364	1,076	1,074	1,074	1,005
PrimMaker	not available	1,225	1,104	1,102	1,102	1,025

Table 9. Number of methods extracted

that CPA is always faster than the 3- and 4-level expansion algorithms (which, as will be demonstrated below, infer less precise types). For the four smallest benchmarks, HelloWorld, Factorial, Richards, and DeltaBlue, the basic and 1-level expansion algorithms are faster than CPA. However, the basic and the 1-level expansion algorithms' precision are close to useless for all benchmarks but the trivial HelloWorld (see below). For most of the benchmarks, the 2-level expansion algorithm is the fastest, although CPA outperforms it on the PrimMaker benchmark. The precision of the 2-level expansion algorithm, however, is never as good as that of CPA (see below).

For the expansion algorithms, two opposing trends battle as the number of expansions increase. On one hand, precision improves, enabling the inferencer to stay clear of “irrelevant” code that is not part of the application. This trend speeds up type inference. On the other hand, redundancy increases, acting to slow down type inference. Beyond 2 levels of expansion, which appears to be the “sweet spot” for the benchmarks in this study (see Figure 30), the effect of the former trend wears off, and the cost of redundancy starts showing through.

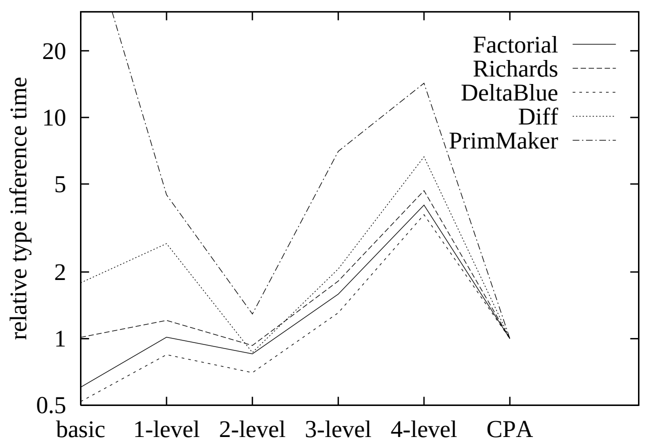


Figure 30. Type inference time (normalized to CPA)

	basic/0-level	1-level	2-level	3-level	4-level	CPA
HelloWorld	0.0	0.0	0.0	0.0	0.0	0.0
Factorial	4.1	6.9	5.8	10.8	27.3	6.8
Richards	7.3	8.7	6.7	13.1	33.6	7.2
DeltaBlue	5.4	8.8	7.3	13.6	37.8	10.4
Diff	27.7	41.7	13.4	32.0	102.8	15.5
PrimMaker	∞	42.9	12.4	67.4	137.1	9.6

Table 10. Type inference time in CPU seconds on a 167 MHz UltraSPARC

Polyvariance is essential to obtain precision, but too much polyvariance can hurt efficiency. Table 11 contains two numbers for each algorithm/program combination: the average number of times each method was analyzed (the number of templates divided by the number of methods), and the average number of times each expression was analyzed. In addition, the second set of numbers is plotted in Figure 31. The first number treats all methods as equally important. However, large methods are more expensive to analyze, so it is particularly important to avoid too much polyvariance for these. The second number, the polyvariance degree of expressions, corrects for this lack of discrimination: it assigns more significance to large methods when computing the averages. For both methods and expressions, the polyvariance degree of CPA is close to that of the 3-level expansion algorithm, yet CPA is faster than the 3-level expansion algorithm because monomorphic templates can be processed faster.

Table 12 quantifies the precision the algorithms on the benchmarks, and Figure 32 illustrates it graphically. The measure of precision used is the average size of the types inferred for all expressions. The averages are over the polyvariant domains, i.e., if an expression was analyzed twice, it counts twice in the average. The orientation of the y-axis in the figure makes a more precise algorithm appear closer to the bottom. The y axis starts at 1.0; this value is close to the theoretical minimum of the average size of inferred types, given that most expressions in programs do not fail, i.e., have non-empty types. To emphasize the difference between the more precise algorithms, which have average sizes of the inferred types in a narrow range towards the bottom of the scale, and to accommodate the wide range of

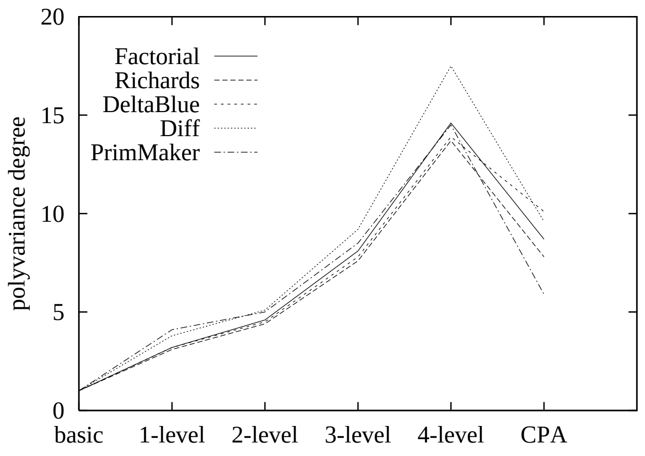


Figure 31. Polyvariance degree for expressions

	basic/0-level	1-level	2-level	3-level	4-level	CPA
HelloWorld	1.0 / 1.0	1.1 / 1.3	1.4 / 1.5	1.6 / 1.7	1.7 / 1.7	1.0 / 1.0
Factorial	1.0 / 1.0	4.1 / 3.2	6.4 / 4.6	11.8 / 8.1	20.6 / 14.6	11.6 / 8.7
Richards	1.0 / 1.0	4.0 / 3.1	6.3 / 4.4	11.6 / 7.6	20.3 / 13.7	10.7 / 7.8
DeltaBlue	1.0 / 1.0	4.0 / 3.2	6.5 / 4.5	11.7 / 7.8	20.2 / 13.9	13.7 / 10.1
Diff	1.0 / 1.0	4.7 / 3.8	7.3 / 5.1	13.7 / 9.2	25.1 / 17.5	12.6 / 9.6
PrimMaker	1.0 / 1.0	4.8 / 4.1	7.3 / 5.0	13.1 / 8.5	22.4 / 14.5	8.3 / 5.9

Table 11. Number of times each method/expression was analyzed (polyvariance degree)

precision exhibited by the algorithms, we used a log scale on the y axis. CPA achieves the lowest sizes, even compared against the 4-level expansion algorithm, which analyze each expression more times (see Figure 31 or Table 11). The expansion algorithms' improvements from level 0 to level 2 are particularly clear both in Table 12 and Figure 32. From level 0 to level 1, the main improvement comes from the 1-level expansion algorithm's ability to analyze two-armed conditional statements precisely (`ifTrue:False:`). Subsequently, the precision increase from level 1 to level 2 is ensured by the 2-level expansion algorithm's ability to analyze one-armed conditional statements precisely (`ifTrue:`); see Section 3.2.3.1. The frequent occurrence of conditional statements, rather than their structure, make them an important component in all the benchmark programs.

	basic/0-level	1-level	2-level	3-level	4-level	CPA
HelloWorld	1.00	1.00	1.00	1.00	1.00	1.00
Factorial	14.68	6.67	1.70	1.66	1.68	1.02
Richards	17.53	9.27	1.84	1.77	1.77	1.02
DeltaBlue	16.53	8.20	1.90	1.81	1.82	1.07
Diff	28.21	15.27	2.10	1.99	1.97	1.16
PrimMaker	not available	17.20	2.36	2.25	2.22	1.19

Table 12. Average size of types for all expressions

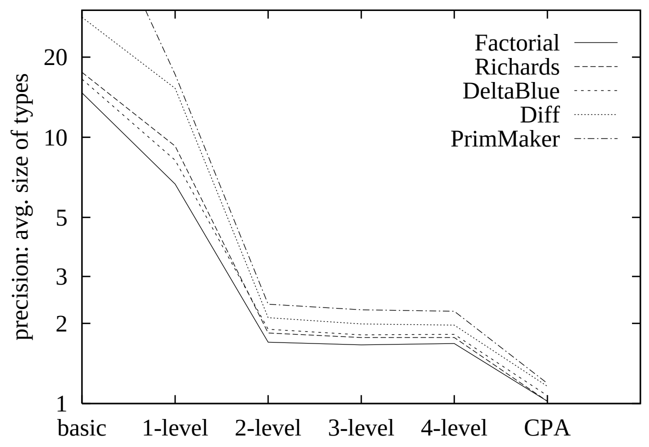


Figure 32. Precision (average size of inferred types) of the type inference algorithms

A visual summary of the performance of the type-inference algorithms can be obtained by plotting the data for each algorithm in a coordinate system where the x-axis indicates speed of type inference and the y-axis precision of type inference. We use type-inference time measured in CPU seconds as the speed indicator (obtained from Table 10) and average size of the inferred types as the precision indicator (obtained from Table 12). Precision is again plotted on a log scale. To avoid clutter, we plot each benchmark in its own coordinate system. Figure 33 shows the resulting graphs. The graphs clearly demonstrate the lack of precision of the basic and 1-level algorithm, the significant precision improvement and cost reduction obtained by the 2-level algorithm, and the exponential increase in cost as the number of expansions increases beyond 2.

To conclude the empirical results, it is amusing to see the types inferred by each algorithm for the expression `50 factorial`. As shown in the introduction of this chapter, the basic algorithm inferred:

$\{\overline{\text{smallInt}}, \overline{\text{bigInt}}, \overline{\text{collector}}, \overline{\text{memory}}, \overline{\text{memoryState}}, \overline{\text{byteVector}}, \overline{\text{mutableString}}, \overline{\text{immutableString}}, \overline{\text{float}}, \overline{\text{link}}, \overline{\text{list}}, \overline{\text{primitiveFailedError}}, \overline{\text{time}}, \overline{\text{vector}}, \overline{\text{sequence}}, \overline{\text{false}}, \overline{\text{true}}, \overline{\text{nil}}, [\overline{\text{blk}}_1], [\overline{\text{blk}}_2], \ldots, [\overline{\text{blk}}_{13}]\}$.

The 1-level expansion algorithm did slightly better:

$\{\overline{\text{smallInt}}, \overline{\text{bigInt}}, \overline{\text{collector}}, \overline{\text{memory}}, \overline{\text{memoryState}}, \overline{\text{byteVector}}, \overline{\text{mutableString}}, \overline{\text{immutableString}}, \overline{\text{float}}, \overline{\text{link}}, \overline{\text{list}}, \overline{\text{primitiveFailedError}}, \overline{\text{time}}, \overline{\text{vector}}, \overline{\text{sequence}}, \overline{\text{false}}, \overline{\text{true}}, \overline{\text{nil}}\}$.

Not surprisingly, the 2-level expansion algorithm was significantly better, although still not perfect:

$\{\overline{\text{smallInt}}, \overline{\text{bigInt}}, \overline{\text{float}}, \overline{\text{primitiveFailedError}}\}$.

Both the 3- and 4-level algorithms inferred the same type as the 2-level algorithm. Finally, CPA inferred:

$\{\overline{\text{smallInt}}, \overline{\text{bigInt}}\}$.

This example completes our discussion of type-inference algorithms for analyzing code with parametric polymorphism. The next chapter, following the summary below, broadens the discussion by taking up several issues that must be considered to implement a comprehensive type-inference system.

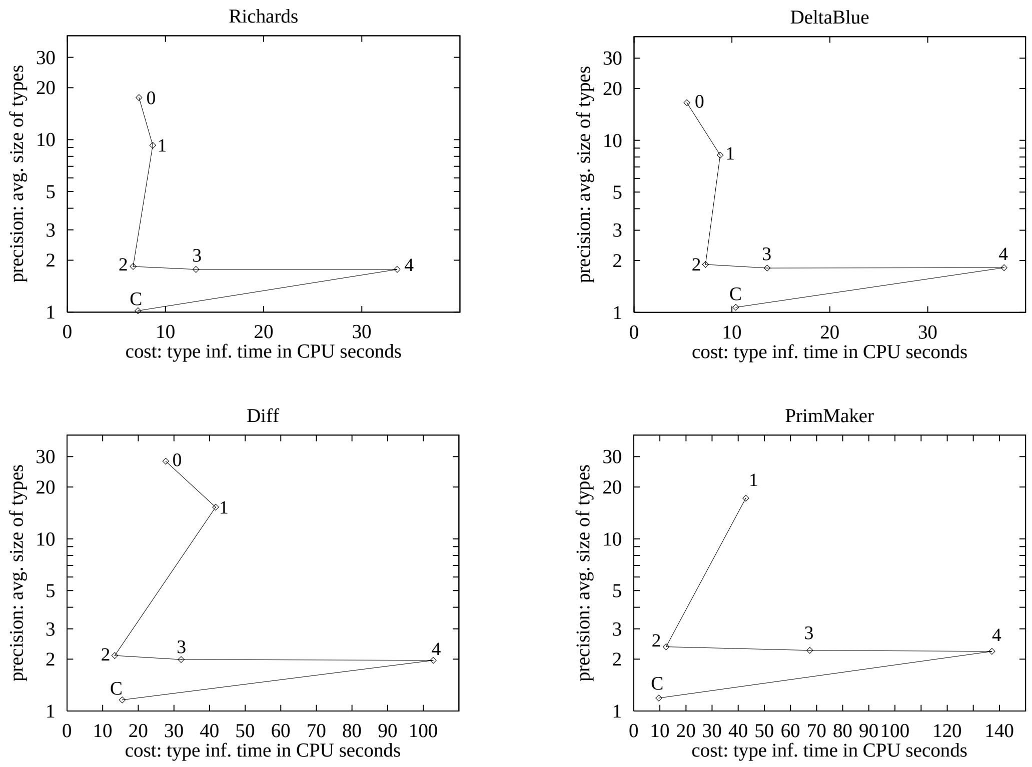


Figure 33. 2D plot of precision and speed of type-inference algorithms

The tick marks are labelled as follows:
0 = basic algorithm, 1 = 1-level expansion, …, 4 = 4-level expansion, C = CPA.

3.5 Summary

Types and programs are defined in terms of objects. A program consists of a main object and a main method in the context of an image of objects; to execute the program, the main method is invoked. To obtain a tractable notion of type, two abstractions are necessary. First, objects are grouped according to structural similarity. Second, groups are closed under cloning to avoid differentiating between an object and its clones. The two abstraction functions commute; the composite function maps objects to object types. Object types are similar to classes, but more general. Finally, types are defined as sets of object types.

With this notion of type, the basic type-inference algorithm can be developed as a flow analysis. The idea is to use constraints to capture all possible data flows during execution of the target program. From observing the myriad of constraints generated during type inference, we derive the need for a higher level of abstraction. Our proposal, *templates*, encapsulates and abstracts individual constraints the same way that procedures encapsulate and abstract individual expressions. Templates permit a concise description of the monovariant basic algorithm and five polyvariant improvements that aim to analyze code with parametric polymorphism more precisely. The uniform and high-level presentation made possible by templates enabled for the first time direct and intuitive comparisons of the algorithms.

We introduced the class of *adaptive algorithms* by means of an example, the ideal algorithm. Adaptive algorithms employ partial type information during inference to improve the cost/precision trade-off. Three specific adaptive algorithms were reviewed: the hash algorithm (by Palsberg, Schwartzbach, and the present author), the iterative algorithm which we viewed as an approximation of the ideal algorithm (by Plevyak and Chien), and the cartesian product algorithm (by the present author). The review of the iterative algorithm focused on parametric polymorphism; in their work, Plevyak and Chien devoted considerable attention to data polymorphism, a problem that we discuss in Section 4.3.

Finally, using the insight gained from studying the previous algorithms, we developed the *cartesian product algorithm*, a new polyvariant adaptive type-inference algorithm. It was motivated by the observation that although send sites can be polymorphic, during execution each specific message carries a monomorphic tuple of arguments. The cartesian product algorithm simulates this behavior, applying the cartesian product to break the analysis of each polymorphic send into a case analysis with each case representing a monomorphic combination of actual arguments. The cartesian product algorithm has several desirable features:

- It is *precise* since it closely mirrors run time by creating templates with monomorphic argument types.
- It is *efficient*, since it performs no redundant work and need not iterate.
- It is *conceptually simple*, involving no program expansions or mapping of type information across iterations.
- It is *general*, permitting a wider range of inheritance schemes than algorithms that need to expand away inheritance (we discuss this property in Section 4.2).

Table 13 condenses the most important attributes of the six algorithms discussed. The efficiency column is based on the informal remarks made when the algorithms were presented. In Section 3.4, we gave measurements for the expansion algorithms and the cartesian product algorithm.

Algorithm	**Polyvariant?**	**Adaptive?**	**Efficiency**	**Precision**	**Advantage/contrib.**	**Drawback**
basic	no	no	“fast”	inadequate	simple, first	imprecise
1-level expansion	yes	no	slow	meager	first polyvariant algorithm	fails on polymorphic call chains of length >1
p-level expansion	yes	no	slow	good	parameterized cost/ precision trade-off	slow before getting precise (redundancy)
hash	yes	yes	fast	good	good handling of polymorphic receivers	polymorphic arg’s need ad-hoc rules
iterative[a]	yes	yes	medium (?)	better	better cost/precision trade-off	iteration overhead
cartesian product	yes	yes	fast	better still	non-iterative; need not partition sends	scaling?

Table 13. Summary of the six type-inference algorithms

a. Not implemented for Self.

4 From an algorithm to a type-inference system

The previous chapter explored constraint-based type inference focusing on how different algorithms analyze code with parametric polymorphism. However, a complete type-inference system involves more than an algorithm for analyzing parametric polymorphism. This chapter covers several issues that arose while designing and implementing the Self type inferencer. The issues apply to a wider range of systems than just Self-like languages or environments. The chapter starts with issues that address specific language features and then moves on to issues that relate to environment properties. The following list briefly introduces the issues:

- *Resolution of dynamic dispatch.* Object-oriented languages derive significant expressive power from dynamic dispatch (or late binding), but it is also a special challenge for type inference: in Section 3.2.1.1, we observed the need to resolve dynamic dispatch, but deferred dealing with it. In Section 4.1, we do so and in the process separate three issues that have been tangled in previous work: generation of constraints, resolution of dynamic dispatch, and the role of inheritance.
- *Inheritance.* An inherited method can be invoked with several kinds of receivers. This variability may reduce precision of type inference. Section 4.2 reviews a commonly adopted solution to this problem, copy-down elimination of inheritance prior to type inference, and points out its limitations. Then it explains how the cartesian product algorithm needs no inheritance elimination to boost precision.
- *Data polymorphism.* The previous chapter presented six increasingly precise algorithms for analyzing code with parametric polymorphism. Data polymorphism, while not targeted in our work, can also be a source of imprecision. Section 4.3 reviews other researchers' proposals for analyzing code with data polymorphism, and identifies possible directions for future work.
- *Blocks (closures) and block structure.* Self's blocks, like Smalltalk's, can access variables in their lexical scopes and perform non-local returns. Section 4.4 describes how to analyze both features precisely and efficiently. Our approach applies to block structure in general, including Scheme's closures, Pascal's nested procedures, and Beta's nested patterns.
- *Flow-sensitive analysis of variable accesses.* Adding a type system to an existing body of code may require the additional flexibility and precision permitted by inferring a distinct type for each variable *access* (versus inferring one type for each *variable)*. Section 4.5 presents within the framework of constraint-based analysis four increasingly powerful systems for flow-sensitive analysis of variable accesses.
- *Grouping.* In the Self programming environment, objects are primary. While source code modules can be derived from objects, the modules do not necessarily capture the full state of the image, e.g., there may be objects that belong to no module. Consequently, we have chosen to infer types directly on objects. Section 4.6 discusses issues arising from this choice.
- *Incremental type inference.* Some applications of type information demand fast type inference. Section 4.7 proposes a simple incremental analysis technique. The incremental analyzer can modify previously inferred types to account for certain changes in the target program faster than the non-incremental analyzer can re-analyze the modified program. Consequently, type inference of a sequence of similar programs can be performed faster, as can analysis of programs sharing "code libraries."

The issues above pertain regardless of which of the six inference algorithms from the previous chapter is used; e.g., all algorithms need to resolve dynamic dispatch. However, as will be explained, some of the issues have interesting interactions with specific algorithms. While no issue is exclusive to Self, they need not all apply to a given language; e.g., C has no block structure, rendering the block structure issue irrelevant.

4.1 Dynamic dispatch resolution (lookup)

Recall that Section 3.2.1.1 showed how to set up constraints for different kinds of *targets* (e.g., methods and assignable slots) that dynamically-dispatched message sends may "invoke." However, one problem was deferred: *resolving* *dynamic dispatch* to find the targets in the first place. This section tackles dynamic dispatch resolution. More precisely, given a send, we show how to find a conservative and precise approximation to the set of targets it may invoke:

- The approximation must be conservative, i.e., include all the possible targets that can occur at run-time, to ensure correctness of type inference.
- The approximation must be precise, since some object-oriented programs use dynamically-dispatched sends frequently. An unnecessary precision loss at each send, however minor, may affect overall precision significantly when the losses are accumulated over an entire program†. This is especially true for Self programs, since they use dynamically-dispatched sends more frequently than programs written in other object-oriented languages.

Some of the previous work on static analysis of object-oriented programs made very conservative assumptions about the targets of dynamically-dispatched sends. For example, the type inferencer for Smalltalk-76 that Suzuki describes in [111] (see also Section 2.3.2.3) *starts* with the assumption that a message may invoke *any* method with a matching selector. While this approach may have worked in the limited Smalltalk-76 system, it does not scale well: when a system grows, the number of incidental name clashes increases, in turn decreasing the precision of the initial dispatch resolution. Consequently, for Self, and probably also today's Smalltalk, this dispatch resolution scheme cannot support precise type inference. In particular, sends with common selectors such as `value` cannot be analyzed precisely because an extremely large number of objects implement `value`‡.

In a statically-typed language such as C++, one could use the receiver's static type (class) declaration to approximate the set of methods that may be invoked. Südholt and Steigner took this approach [113], but give no empirical results. Resolution based on the static type yields better precision than that of Suzuki's initial dispatch resolution: the static type narrows the set of targets from all methods with a matching selector to those defined in subclasses of the statically declared class. However, the resolution may still not be sufficiently precise. In particular, if the programmer strives to write reusable code, i.e., uses the weakest possible static type declarations to retain maximal polymorphism, the static type declarations convey little information and therefore cannot support precise dispatch resolution. Moreover, reliance on static type declarations may create a situation that encourages the programmer to write *less* reusable code. ("It executes faster, so why not?")

Palsberg and Schwartzbach had the key insight that the type inferred for a send's receiver could be used to limit the set of targets that need to be considered [86]. For example, if the type of `x` in `x+y` is $\{\overline{\text{smallInt}}\}$, the type inferencer can safely ignore the `+` method found in bigInts, since it will not be invoked. In the presentation given in [86], resolution of dynamic dispatch was not separated from setting up the constraints that capture the semantics of invoking the target methods. Instead, the combination of the two problems was modelled using a collection of *conditional constraints,* formulas of the form:

$$c \in \text{type}(R) \Rightarrow \text{type}(A) \subseteq \text{type}(F)$$

Without going into details, the constraints on the right hand side of the implication arrows, $\text{type}(A) \subseteq \text{type}(F)$, capture the semantics of method invocations, i.e., reflect data flows like the (unconditional) constraints we have used throughout this and the previous chapter. The reader may think of *A* as an actual argument of a send and *F* as a formal argument of a possible target for the send. The type-inference algorithm in [86] connects every send with a given selector to *all* methods with that selector, i.e., it generates a superset of the constraints that our algorithm generates. It balances this excessive constraint generation by the conditions on the left hand side of the implication arrows, $c \in \text{type}(R)$. The conditions encode degenerate lookup rules (simplified to reflect that inheritance has been expanded away), ensuring that only the constraints describing invocations of targets permitted by the type information have an effect. The reader may think of *c* as a particular class and *R* as the receiver expression of a send; only if the class *c* belongs to the type of *R* can the method with a matching selector in class *c* be invoked by the given send.

† In general, some conservatism is necessary to avoid incomputability, of course.

‡ The system described in [111] iterates type inference to improve on the initial assumptions and gain precision. Even so, the inferencer must get through the first very pessimistic iteration before the ball can start rolling. We have more to say about iteration in Section 4.1.3 where we characterize the present problem of the initial iteration as the "initial hurdle."

Conditional constraints model the combination of dynamic dispatch resolution and the effect of method invocations. Thus, they would seem like a perfect tool for analyzing object-oriented languages. However, this failure to separate concerns also makes them less flexible. In [86] and [85], type inference took place by invoking an algorithm specifically designed to solve collections of constraints of the form:

$$c \in \text{type}(R) \Rightarrow \text{type}(A) \subseteq \text{type}(F) .$$

When the present author worked with Palsberg and Schwartzbach to generalize this system from idealized Smalltalk to Self [6], which have roughly the same method invocation semantics, we found it impossible to express the lookup rules of Self with simple conditions of the form $c \in \text{type}(R)$. First, dynamic inheritance could not be expanded away, making it necessary to express lookup rules as sequences of inclusion conditions describing parent chains. Second, multiple and cyclic inheritance meant that such sequences of conditions had to be generated for a countably infinite collection of possible parent chains, rather than a single linear chain. Consequently, the existing constraint solving algorithm could no longer be used. Rather than attempting to devise a more powerful constraint solver for the more complicated conditional constraints, an alternative approach was taken. The alternative approach completely separates dispatch resolution from analysis of the identified target methods. Moreover, it directly employs the lookup algorithm during type inference, instead of indirectly encoding lookup rules as boolean conditions.

The alternative approach takes advantage of the fact that the inferred types of receiver expressions can be used to resolve dynamic dispatch incrementally as type inference proceeds. The apparent circularity, that types are needed to resolve dynamic dispatch but dynamic dispatch must be resolved to infer types, can be broken by exploiting the monotonicity property (i.e., that types are monotonically growing sets of object types during type inference)†. To see how this works, let *TS* denote the set of targets for a send expression such as:

```
    rcvrExp foo.
```

(The above expression sends the message `foo` to the object that results from evaluating the receiver expression `rcvrExp`, which may itself be a dynamically-dispatched send expression.) While the set of targets for the `foo` send expression cannot (and need not) be computed *in advance*, it can be computed incrementally during type inference. Assume that *TS* is initialized to the empty set, reflecting that type(`rcvrExp`) is empty initially (if a receiver expression has the empty type, the send cannot invoke any targets). We extend the type inferencer so that each time it propagates an object type into type(`rcvrExp`) it performs a lookup of `foo` starting in the object type. Any targets that the lookup finds are added to *TS*. In addition to adding the targets to *TS*, the type inferencer establishes constraints as described in Section 3.2.1.1 to capture the effects of invoking these targets. At the completion of type inference, not only does type(`rcvrExp`) hold a sound type for the receiver expression, *TS* contains the corresponding set of targets:

- *TS* will contain no excess targets. Monotonicity ensures this: an object type that is added to a receiver type will never be removed again. Thus, any targets that were found when the object type was added are still required at completion of type inference.

- *TS* will contain all necessary targets. This follows trivially since it is impossible to add an object type to a receiver type without also checking for targets.

The incremental computation of target sets employs the normal run-time lookup algorithm directly during type inference. Therefore, adapting type inference to a new language with different lookup rules (be it multiple dispatch, multiple inheritance with an exotic disambiguation rule, or something else) is as simple as plugging in a new lookup algorithm. The rest of the type inferencer is fully isolated from the lookup rules of the analyzed language. Of the type-inference algorithms surveyed in Section 2.3.2, several attain a similar encapsulation of the lookup rules by virtue of expanding away inheritance before inference starts: only the expander depends on the lookup rules. However, as pointed out above, dynamic inheritance cannot be eliminated by an expansion, and even when an expansion is possible, this approach is not as direct as employing the lookup algorithm during type inference (essentially,

† In Section 3.2.8.2, a similar problem was encountered and solved: the cartesian product algorithm needs to use types to guide sharing while it is in the process of computing these types. The solution for guiding sharing parallels the solution adopted here for dispatch resolution: incremental computation.

expansion computes the inverse function of lookup). For these reasons, we prefer the approach of performing lookups incrementally during type inference.

Strictly speaking, the exact run-time lookup algorithm cannot be used in the Self type inferencer. This algorithm operates on objects, but at type-inference time only clone families are available. However, except for dynamic inheritance, all members of a clone family have the same contents in their parent slots, so the difference is immaterial. The one case where the difference matters, dynamic inheritance, is addressed below.

4.1.1 Dynamic inheritance

Dynamic inheritance, while adding significant expressive power to Self, also poses an additional challenge to type inference by making dynamic dispatch resolution harder. Even when the exact receiver type of a send is known, dynamic inheritance may still obscure which method(s) may be invoked.

Figure 34 shows how dynamic inheritance can be used to implement binary search trees. In the Self system, such search trees are used to implement both ordered sets and bags (multi-sets). However, to keep the example simple, here we have removed a level of inheritance that factors out the commonality between sets and bags. In this example, dynamic inheritance gives `treeNode` objects modal behavior: a node in a binary tree behaves differently depending on whether the subtree it roots represents the empty collection (the node is a leaf) or a non-empty collection (the node is an interior node). For example, when a `treeNode` receives the `includesKey:` message to test whether the collection represented by its subtree contains a given key, the dynamic parent in `treeNode` selects one of two implementations:

- In the leaf mode, the message invokes the `includesKey:` method in `emptyNode`. This method, while not shown on the figure, simply returns `false`.
- In the interior mode, the message invokes the `includesKey:` method in `nonEmptyNode`. This method, also omitted from the figure, compares the argument against the key in the current node; if they are equal, it returns `true`; otherwise, it recursively sends `includesKey:` to the left or right subtree.

Consider now dynamic dispatch resolution. Even if the type inferencer can determine that the receiver expression, `tree`, of this send expression

```
    tree includesKey: 'green'
```

is a `treeNode` object, the dispatch resolver cannot apply the run-time lookup algorithm to determine which method is invoked. If it did, after looking in vain for an `includesKey:` slot in `treeNode`, the run-time lookup algorithm would consult the dynamic parent slot and continue the lookup into the object currently found there. Consequently, it would fail to account for assignments to the dynamic parent slot and therefore resolve the dynamic dispatch unsoundly.

Instead of applying the exact run-time lookup algorithm, a more conservative approach must be used to ensure soundness. The simplest approach, but also the most pessimistic, assumes that *any* object may be stored in *any* dynamic parent slot. Unfortunately, this assumption does not permit precise type inference. Whenever the dispatch resolver encounters a dynamic parent slot during a lookup, precision drops dramatically: any slot with a matching selector becomes a possible target. Thus, for messages whose lookups pass through dynamic parent slots, the resulting dispatch resolution is as pessimistic as the resolution in the initial iteration of Suzuki's algorithm.

To do better, we must avoid making the pessimistic assumption that any object can be stored in any dynamic parent slot. The solution, again, is to use the inferred types, this time to restrict the set of possible objects in dynamic parent slots. When tracing a lookup through a dynamic parent slot, it suffices to consider the objects in the extension of the parent slot's type. For example, if the inferred type for the dynamic parent slot in $\overline{\text{treeNode}}$ is $\{\overline{\text{emptyNode}}, \overline{\text{nonEmptyNode}}\}$, a lookup through the parent slot must continue into both $\overline{\text{emptyNode}}$ and $\overline{\text{nonEmptyNode}}$ (conservatively assuming that both possibilities may occur), but can safely ignore *all* other objects in the image.

Again, a circularity looms: to infer types, the algorithm must trace lookups through dynamic parents slot, but to trace lookups, it needs to know the types. And again, monotonicity and incremental computation comes to the rescue.

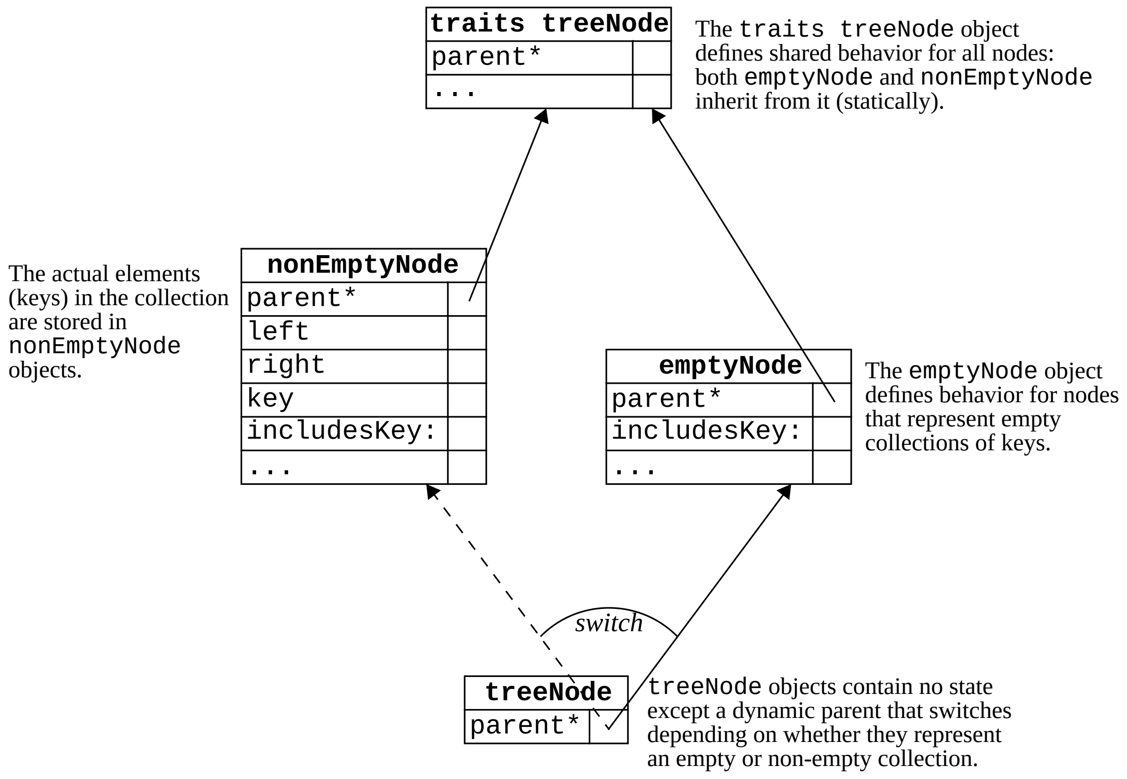

traits treeNode
parent*
...
The traits treeNode object defines shared behavior for all nodes: both emptyNode and nonEmptyNode inherit from it (statically).
nonEmptyNode
parent*
left
right
key
includesKey:
...
The actual elements (keys) in the collection are stored in nonEmptyNode objects.
emptyNode
parent*
includesKey:
...
The emptyNode object defines behavior for nodes that represent empty collections of keys.
switch
treeNode
parent*
treeNode objects contain no state except a dynamic parent that switches depending on whether they represent an empty or non-empty collection.

Figure 34. Binary search tree implementation using dynamic inheritance

When a lookup depends on a dynamic parent slot, the *current* approximation for the parent slot's type is used to complete the lookup. Since monotonicity guarantees that the current approximation is a subset of the final type, the lookup (by construction) will find a subset of the targets that would have been found, had the final type been used. The type inferencer records dependencies between lookups and dynamic parent slots so that if a dynamic parent slot's type later grows, the affected lookups can be extended to take the new possible parents into account.

The left frame in Figure 35 shows the resulting type-inference-time lookup algorithm. To emphasize how it closely parallels the run-time lookup algorithm, the right frame in the figure shows the run-time lookup algorithm. For simplicity, both lookup algorithms ignore the detection of cyclic inheritance and multiple inheritance ambiguities. The normal lookup algorithm computes over the domain of objects and returns a single target or a message-not-understood indicator, whereas the derived algorithm computes over the domain of object types and (conservatively) returns a set of possible targets. Overall, the two algorithms are very similar, an indication that our way of resolving dynamic dispatch is robust, even in the presence of dynamic inheritance, and can generalize to other languages.

This concludes our discussion of dynamic dispatch resolution. The next section shows how to use the knowledge gained from dispatch resolution to strengthen the constraints that describe method invocations.

4.1.2 Receiver constraints

From one viewpoint ("the sender's perspective"), dynamic dispatch *selects* the appropriate implementation of an operation for a given kind of receiver. From a dual viewpoint ("the invoked method's perspective"), dynamic dispatch *filters* the kinds of objects that may be a receiver when a given method executes. For example, consider the methods that implement addition for different kinds of numbers in the Self system. Figure 36 depicts part of the hierarchy for implementing smallInts and bigInts. All smallInts inherit from the object `traits smallInt`, which defines, among many other methods, an addition method, `+`. Similarly, all bigInts inherit from `traits bigInt`,

```
Procedure TI_Lookup(ot,sel)
var
  ts, c
begin
  if "ot has sel slot" then
    return {ot}
  end;
  ts := {};
  forall p in ParentSlots(ot) do
    forall c in type(p) do
      ts := ts ∪ TI_Lookup(c,sel)
    end
  end;
  return ts
end TI_Lookup.
```

```
Procedure Lookup(obj,sel)
var
  ts, c
begin
  if "obj has sel slot" then
    return {obj}
  end;
  ts := {};
  forall p in ParentSlots(obj) do
    c := contents(p);
    ts := ts ∪ Lookup(c,sel);
  end;
  if ts = {} then
    return msgNotUnderstood
  end;
  return someMember(ts)
end Lookup.
```

Figure 35. Lookup algorithms for type-inference time and run time

The right frame shows a simplified version of Self's lookup algorithm (it ignores testing for cyclic inheritance structures and ambiguous sends). The left frame shows the derived lookup algorithm that the type-inference algorithm employs. The algorithms have parallel structure. The main difference is that the run-time lookup algorithm returns a single target, whereas the type-inference version returns a set of possible targets.

which defines the addition method for bigInts. (For simplicity, here we ignore that arithmetic employs double-dispatching in the Self system.)

To avoid ambiguity, we use notation like $+_{\texttt{smallInt}}$ and $+_{\texttt{bigInt}}$ in the following. Consider now a send, `x+y`, for which

$$\text{type}(\texttt{x}) = \text{type}(\texttt{y}) = \{\overline{\text{smallInt}}, \overline{\text{bigInt}}\}.$$

The dynamic dispatch resolver will determine that the possible targets are:

$$TS = \{+_{\texttt{smallInt}}, +_{\texttt{bigInt}}\}.$$

Thus, constraints must be set up between the send and templates for the two + methods to capture these possible invocations. For the argument, the constraints are straightforward, since dynamic dispatch is not an issue:

$$\text{type}(\texttt{y}) \subseteq \text{type}(\texttt{arg}_{\texttt{smallInt}})$$
$$\text{type}(\texttt{y}) \subseteq \text{type}(\texttt{arg}_{\texttt{bigInt}})$$

($\texttt{arg}_{\texttt{smallInt}}$ and $\texttt{arg}_{\texttt{bigInt}}$ denote the formal arguments of $+_{\texttt{smallInt}}$ and $+_{\texttt{bigInt}}$, respectively.) However, for the receiver, constraints like

$$\text{type}(\texttt{x}) \subseteq \text{type}(\texttt{self}_{\texttt{smallInt}})$$
$$\text{type}(\texttt{x}) \subseteq \text{type}(\texttt{self}_{\texttt{bigInt}})$$

are not optimal. According to the filtering perspective on dynamic dispatch, $+_{\texttt{smallInt}}$ should be invoked on smallInt receivers only, and $+_{\texttt{bigInt}}$ on bigInt receivers only. To avoid losing precision, the type inferencer must respect this restriction. In general, before propagating the type of a receiver expression into the type of `self` in a target method, the receiver type must be filtered: only object types for which a lookup may lead to the given target should be propagated into the type of `self` in that target. Figure 37 illustrates this graphically.

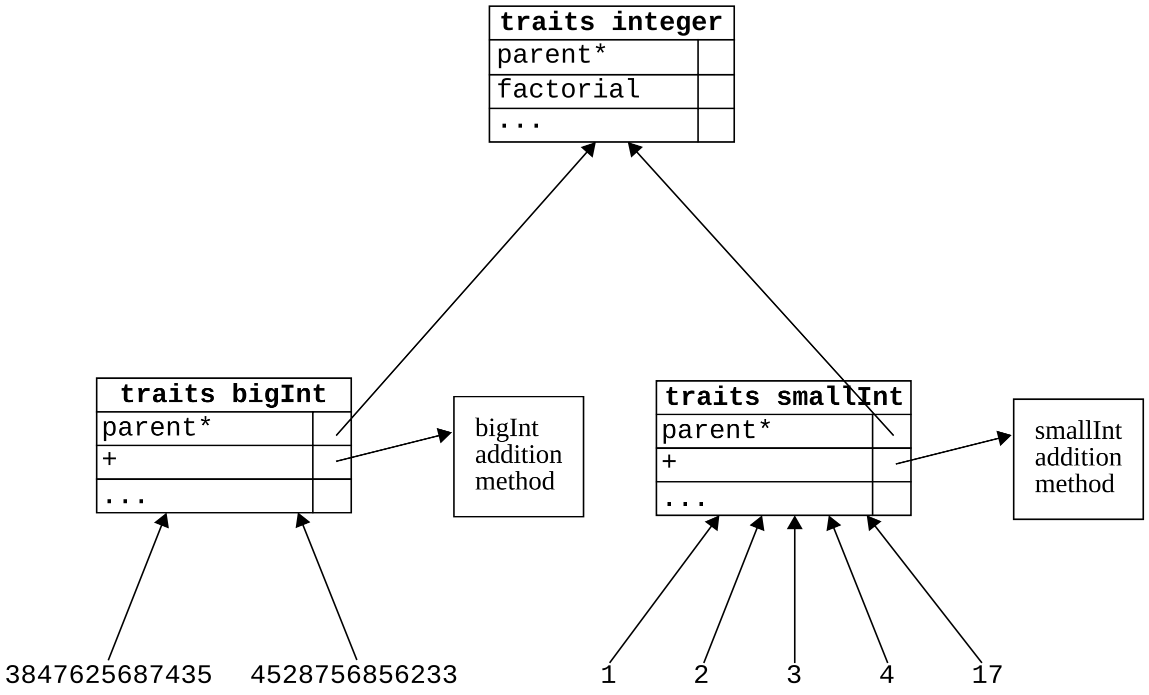


Figure 36. Part of the number hierarchy in the Self system

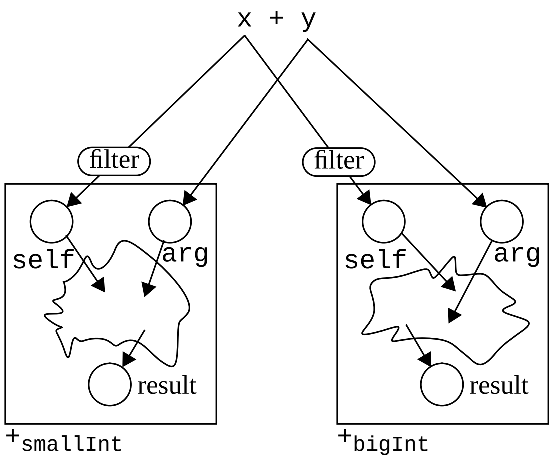


Figure 37. Viewing dynamic dispatch as a restriction on the receiver

Given a method, dynamic dispatch can be viewed as a restriction on which objects the method may be invoked upon: only objects from which a lookup can find the method can be a receiver. The type inferencer captures this restriction by inserting filters between receiver expression types and `self` types. Above, the filters prevent bigInts from reaching `self` in the smallInt addition method and vice versa.

The filters can be implemented by a simple extension to the dynamic dispatch resolution code. Instead of just recording targets in *TS*, the dispatch resolver records pairs, (*M*, ρ):

- *M* is a possible target as before.
- ρ is a receiver that leads to the target *M*.

Thus, the dispatch resolver records both the possible targets and the specific receiver object types leading to them. For instance, in the `x+y` example, it records:

$$TS = \{(+_{\texttt{smallInt}}, \overline{\text{smallInt}}), (+_{\texttt{bigInt}}, \overline{\text{bigInt}})\}.$$

In general, the set of targets may contain several pairs with the same first component, but different second components. This can happen when a method is inherited by several kinds of objects; e.g., a `factorial` method defined in `traits integer` and inherited by both smallInts and bigInts may appear twice: {(`factorial`, $\overline{\text{smallInt}}$), (`factorial`, $\overline{\text{bigInt}}$)}.

With the dynamic dispatch resolver extended to compute sets of pairs, it is easy to implement the filter that determines whether a given receiver object type ρ should propagate into the type of `self` in a target method *M:*

$$\text{propagate} \Leftrightarrow (M, \rho) \in TS$$

We have now completed our study of how to analyze dynamic dispatch in a constraint-based framework. Before proceeding to the next issue, inheritance, we discuss two general solution techniques that are illustrated by dynamic dispatch resolution and will be a recurring theme.

4.1.3 Iteration or integration?

Dynamic dispatch resolution and type inference depend on each other: to solve either problem, the solution to the other must be at least partially known. We will later encounter other mutually dependent problems. Thus, it is worth taking a moment now to consider two general solution techniques: iteration and integration. We present the techniques in terms of two problems, type inference and "another problem," but it is straightforward to generalize to three or more inter-dependent problems. Moreover, the issue is not specific to type inference.

- *Iteration.* The two problems can be solved sequentially and the process iterated to gain precision: type inference, "other analysis," better type inference, better "other analysis," even better type inference, etc. The initial type inference step employs minimal (or trivially true) assumptions, but in subsequent iterations the information computed in the previous iteration is available to establish more precise assumptions. A successful iteration scheme should converge in few rounds and the limit value should be precise.

  Example. Suzuki's system resolves dynamic dispatch iteratively (see [111] and Section 2.3.2.3): infer types assuming that sends may invoke *any* method with a matching selector, use the inferred types to resolve dynamic dispatch more precisely, infer more precise types, resolve dynamic dispatch even more precisely, etc.

  Example. Suzuki's system infers types for instance and global variables iteratively (see [111] and Section 2.3.2.3), starting with the pessimistic assumption that variables may hold objects of any class, and narrowing these types from iteration to iteration.

  Example. Adaptive type inference is amenable to iteration. The two problems are type inference and the sharing problem (i.e., "is it OK to connect these two sends to a shared template?"—see Section 3.2.5). Plevyak and Chien's iterative algorithm (Section 3.2.7) performs iteration to solve these two problems: infer types using a trivial sharing decision (always share), use the inferred types to find better answers to the sharing questions, infer better types, etc.

- *Integration.* The two problems can be solved simultaneously. To succeed, the analyses must compute information incrementally (to make partial information available to each other) and be able to work with partial information. In particular, for type inference this means that monotonicity must be preserved and that the other analysis must compute with monotonically growing approximations of the final types.

  Example. The dynamic dispatch resolution described in Section 4.1 is integrated with type inference. The initial assumption is that dynamically-dispatched sends can invoke *no* targets. As the type inferencer develops the constraint graph and more and more object types are added to the types of receiver expressions, the dispatch resolution algorithm updates the set of targets incrementally and monotonically (targets are never retracted). As new targets are found, constraints are added (monotonically) and types grow (monotonically).

Example. The Self type inferencer and most of the other ones surveyed in Section 2.3.2 integrate type inference for variables with type inference for other constructs in the language, assuming initially that types of variables are empty and gradually relaxing this assumption as assignments are encountered.

Example. Adaptive type inference is also amenable to integrated solutions: the cartesian product algorithm (Section 3.2.8) is an integrated solution to a modified sharing problem and type inference.

The following comparison of iteration and integration provides a better understanding of their strengths and weaknesses. On the former two aspects of the comparison, iteration comes out ahead. On the latter three aspects, integration has the edge.

- *Expressive power.* Iteration has greater expressive power since it can violate the monotonicity requirement across iterations. For example, Suzuki's dispatch resolution is free to both add or remove targets from one iteration to the next. In contrast, integration must respect monotonicity. If the integrated dispatch resolver attempts to retract a target, it may lead to an inconsistent constraint graph (one that contains constraints for impossible targets)[†].

- *Modularity.* Iteration permits the separation of the two analyses, easing both implementation and reasoning about correctness. Integration does not straightforwardly permit separation. (However, we think that it may be possible to formalize an interface between two incremental analyzers and thereby achieve modularity for certain classes of problems.)

- *Efficiency.* Since integration preserves the fundamental monotonicity requirement of constraint-based analysis, it can execute in one pass and therefore deliver high performance. In contrast, iteration can be costly, since each iteration involves a full type inference.

- *Initial hurdle.* The iterative approach encounters the greatest difficulties in the first iteration when it operates under the most pessimistic assumptions. While this may not always be a problem, sometimes it is. For instance, consider the iterative algorithm (Section 3.2.7) which is equivalent to the basic algorithm (Section 3.2.1) in its first iteration. We observed in Section 3.4 that the basic algorithm is so inaccurate that it keeps computing until memory is exhausted, even when applied to an expression as simple as `50 factorial`. Consequently, the iterative algorithm in its pure form faces a potentially insurmountable initial hurdle. In comparison, integrated analyses face no initial hurdle, since they obtain full precision in the first (and only) iteration.

- *Asymptotic precision*. If the iterative approach overcomes the initial hurdle and furthermore converges, can we rest assured that the precision at the limit is the same as that of the integrated approach? The general answer, unfortunately, is no[‡]. Suzuki's iterative dispatch resolution provides the counter-example: in Section 2.3.2.3 we gave an example where it converges to a less precise result than the integrated dispatch resolution. Intuitively, it is not surprising that the strict sequencing of analyses may decrease precision.

The integration versus iteration issue goes beyond type inference. Wegman and Zadeck identify the issue in the specific setting of constant propagation and dead code elimination [124] (see also Section 2.3.2.8). They observe how many optimizing compilers iterate constant propagation and dead code elimination[††]: when more constants have been found more code is dead (since more conditional tests are constant); when more code is dead more expressions are constant (since fewer assignments remain to introduce variability). Wegman and Zadeck present an integrated algorithm for the constant propagation and dead code elimination problems for a non-object-oriented language and argue that it is more precise than iterating algorithms that solve the separate problems. In our terminology: Wegman and Zadeck's integrated algorithm surpasses the asymptotic precision of the iterative combination. Wegman and

[†] If a target is retracted, the type inferencer cannot remove the constraints that were added to the constraint graph when the target was first reported. Thus, the constraint graph will contain *extra* constraints, i.e., the inferred types are less precise, but still sound. To stay within the framework of constraint-based analysis, here we refrain from considering more extensive "repair" work upon performing a non-monotonic operation on the constraint network. However, the artificial intelligence literature describes "truth maintenance systems," which demonstrate the viability of such repair work.

[‡] This observation does not rule out that precision may be the same for specific analyses.

[††] A well-known problem for compiler writers: the phase-ordering problem [44, 45, 47].

Zadeck characterize iterative algorithms as pessimistic (the initial hurdle) and integrated algorithms as optimistic (until they finish their computation, the intermediate results are too optimistic).

The choice between iteration and integration crystallized in our work from combining solutions to the problems of dynamic dispatch resolution and type inference. In the literature, we found it discussed in the setting of constant propagation. The problem is general and has significance beyond these particular analyses, and it will surface again in other contexts later in this chapter.

4.2 Inheritance

Inheritance plays a central role in object-oriented programming [125]. It allows structuring of code to factor out similarities between different kinds of objects: a method defined in a class *C* applies not only to instances of *C*, but also to instances of subclasses of *C*[†]. For example, consider the situation illustrated in the left half of Figure 38. A `display` method is defined in the class `Point` and inherited by the subclass `ColorPoint`. The `display` method can therefore be invoked on both `Point` and `ColorPoint` instances.

When a method is inherited, the types of many of its expressions may change. Comparing an invocation of `display` on a `Point` object and a `ColorPoint` object, in the former the expression `self` has type {`Point`}, whereas in the latter it has type {`ColorPoint`}. In turn, any message that `display` sends to `self` may resolve to different targets, again resulting in different types. Inheritance may—by convention or enforcement—be used in a way that attains substitutability, i.e., ensures that a subclass instance can be used wherever a superclass instance is expected. Even so, the fact remains that types of expressions may change when methods are inherited. Types that change leave us with two choices for type inference:

- The inferencer can compute types that are general (hence: imprecise) enough to cover all cases.
- The inferencer can differentiate between the different cases to compute more precise types for each one.

To obtain better precision, many researchers have opted for the second alternative. Commonly, they realize it by applying a preprocessor to eliminate inheritance prior to type inference [48, 49, 86, 120]. The preprocessor copies methods down into the classes inheriting them. The result of this *inheritance expansion* is shown in the right half of Figure 38: there are now two identical methods $\texttt{display}_{\texttt{Point}}$ and $\texttt{display}_{\texttt{ColorPoint}}$. The former applies to points only, whereas the latter applies to colored points only.

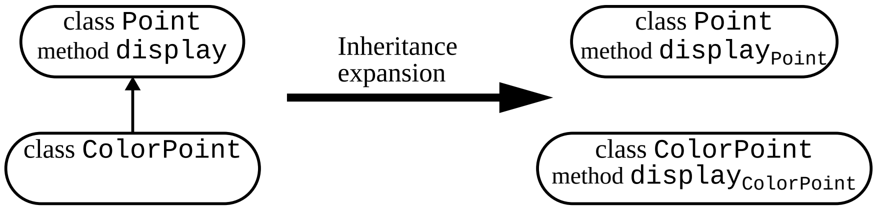


Figure 38. Inheritance expansion

Most type-inference algorithms expand away inheritance to gain precision. The expansion copies inherited methods down into subclasses, allowing expressions in the original method to have several different types, depending on the class of the receiver.

The inheritance expansion improves precision because inaccuracies resulting from analyzing a single method in the context of several classes are avoided. For example, two types will now be inferred for each expression in the original `display` method: one type in $\texttt{display}_{\texttt{Point}}$, valid for invocations of `display` on points, and one in

[†] In Self, which has no classes, inheritance is a relation between objects: inherited methods can be applied to any object that inherits from the object containing the method. To use terminology with which most readers are familiar, we give examples from class-based languages in this section. The fundamental issues, however, remain the same whether or not the language has classes.

$\texttt{display}_{\texttt{ColoredPoint}}$, valid for invocations on colored points. Compiler writers facing the same problem found a similar solution: customization [28], the recompilation of methods for each class inheriting them, making it possible to generate more specific and therefore more efficient code.

The inheritance expansion can be expensive. In the worst case, the program size increases quadratically, even for programs using only single inheritance [86]. Care must also be taken to handle overridden methods correctly, including sends to `super` ("resends"). For Self, there are additional complications: an expansion cannot eliminate inheritance of state and dynamic inheritance. The next section explains how the cartesian product algorithm avoids these complications without losing precision.

4.2.1 Inheritance and the cartesian product algorithm

Does the cartesian product algorithm need an inheritance expansion to boost precision? It does not; it can analyze an unexpanded program with the same precision as an expanded one. Unlike other algorithms, CPA always creates templates in which the type of `self` has exactly one member, i.e., CPA always analyzes a method in the context of one class at a time[†]. Avoiding the inheritance expansion has several advantages:

- *Simplicity.* CPA reuses an existing mechanism, the cartesian product, instead of applying a special-purpose expansion to deal with inheritance.
- *Efficiency.* Templates are created only as needed, whereas the static expansion assumes that a method can be invoked on any class inheriting it. (The expansion could be optimized, of course.)
- *Generality.* CPA handles inheritance of state, dynamic inheritance, and multiple inheritance. It furthermore supports languages with multiple dispatch (multi-methods) [15, 27], by a straightforward extension to the singly-dispatched case: each tuple in the cartesian product is propagated to a template for the *specific* method that implements the operation for the combination of arguments in the tuple.

The next section further explores the relationship between inheritance expansion and the cartesian product algorithm.

4.2.2 From inheritance expansion to the cartesian product algorithm

The inheritance expansion was introduced because types of expressions may change when methods are inherited. An alternative view is that types of expressions change when the type of the first argument (i.e., the type of the receiver) changes. But this is not the end of the story. Types of expressions may depend on the type of any argument, be it the receiver or another one. At best, one can argue that messages are sent more often to `self` than to other arguments and that, accordingly, types depend to a higher degree on the receiver than other arguments. This difference can be attributed to a combination of factors:

- *Programming conventions.* Programmers prefer to think of the first argument as special, often placing methods in the "most accessed" class.
- *Single dispatch.* The asymmetry introduced by single dispatch makes it more convenient to use the first argument when performing operations that may be implemented by several different methods.
- *Other language factors.* Self offers a briefer syntax for (most) messages sent to the receiver: the word `self` can be omitted. Smalltalk methods can only access instance variables in the receiver. C++ privacy rules restrict access to private members to those in the receiver.

In a multi-dispatched language, the second and possibly the third factors are irrelevant. Thus, one would expect type dependencies to be spread more evenly among the arguments. In this light, the inheritance expansion controls polymorphism in a particular argument, the receiver. The choice of argument makes sense since the receiver is *often* the

[†] The instance of the hash function algorithm described in Section 3.2.6 shares this property. However, it applies to singly-dispatched languages only.

most important argument to control, but the expansion is only a partial solution: other arguments, while generally less important, *are* sometimes as important. Two examples from Self are:

- The `ifTrue:False:` methods implementing conditional statements (normally) have only two possible receivers, `true` and `false`, but many possible arguments.
- In doubly-dispatched binary operations, including the frequently used arithmetic operations, the receiver and argument are swapped between the dispatches; they are clearly equally polymorphic.

To fully carry through the expansion idea and gain all the precision possible, methods should be copied, not only "down" to each subclass inheriting them, but also "into" each possible class for each argument. This full expansion ensures precision similar to that of CPA, since methods are customized equally for all arguments (one step is missing: the splitting of actual argument types into monomorphic tuples before propagating them to templates for the method copies).

The full expansion ensures high precision, but incurs exponential cost. A k-argument method, found in a program with n classes and inherited by m of the classes would need to be copied mn^{k-1} times: dynamic dispatch restricts the receiver to be an instance of one of the m subclasses that inherit the method, but for the remaining $k-1$ arguments there are no restrictions[†], i.e., each of them can be an instance of any of the n classes in the program. Hence, the full expansion seems infeasible even for medium-sized programs.

Given that the full expansion is infeasible, how can CPA be feasible? The crucial difference is that CPA only creates templates as it encounters sends that may invoke them with a particular combination of arguments. Thus, methods may *a priori* seem extremely polymorphic, but typical programs utilize only a limited amount of this potential polymorphism. Therefore, CPA, and adaptive type-inference algorithms in general, perform well in practice since they adapt to the actual polymorphism in programs, rather than attempting to address the potential polymorphism.

4.3 Data polymorphism

The presentation of the six type-inference algorithms in the previous chapter was structured around the analysis of code with parametric polymorphism, i.e., methods that may be invoked on different kinds of arguments. Data polymorphism refers to the ability of a slot (or variable) to hold different kinds of objects. For example, "link" objects used in lists of integers, floats, and strings exhibit data polymorphism because their "contents" slots contain objects with different object types.

Just as parametric polymorphism may lead to imprecise type inference unless the inferencer avoids merging the differently-typed invocations of methods, data polymorphism may reduce precision unless the inferencer avoids merging the differently-typed instantiations of the objects containing data-polymorphic slots. We have not worked as much on analysis of code with data polymorphism. Our main contribution, the cartesian product algorithm, only addresses parametric polymorphism. It does nothing to improve the analysis of code with data polymorphism, and thus is no better but also no worse in this regard than the basic algorithm.

A number of algorithms that target data polymorphism have been proposed by other researchers. Analogous to analyzing parametric polymorphism by reanalyzing (or copying) methods, these algorithms split object types (or copy classes) in an attempt to keep different uses of (data) polymorphic instance variables separate. Section 4.3.1 extends the definition of object type and associated framework given in Section 3.1 to encompass these algorithms. Following that, Section 4.3.2 concisely describes specific algorithms for analyzing code with data polymorphism. Finally, Section 4.3.3 discuss how our framework may be generalized further, possibly permitting future development of more powerful algorithms.

[†] Unless the language is multi-dispatched or arguments have static type declarations.

4.3.1 Extended framework

The general idea behind the existing algorithms for analyzing code with data polymorphism is that of *splitting* object types (or classes). For example, assume that the image contains two (initial) point objects, both with integer coordinates, that are assigned to the same group:

point = {`protoPoint`, `aPoint`}.

As the names suggest, one of the point objects may be the prototypical point, whereas the other one is less distinguished. Assume now that the target program uses point objects with both integer and float coordinates. To precisely analyze this program, the integer points should be kept separate from the float points to the largest degree possible (the separation may not be possible for all programs or everywhere in a given program, of course). This separation can be achieved by using more than one object type to describe the (infinite) collection of all points that may exist during all executions of the target program. In contrast, the framework given in Section 3.1 assigns the same object type, $\overline{\text{point}}$, to any object cloned directly or indirectly from either `protoPoint` or `aPoint`. Figure 39 illustrates how splitting of object types relates to the definition of object type given in Section 3.1. In the figure, the original point object type, $\overline{\text{point}}$, is split into three object types, $\overline{\text{point}_1}$, $\overline{\text{point}_2}$, and $\overline{\text{point}_3}$. For specificity, we denote the object types resulting from splitting *refined object types.*

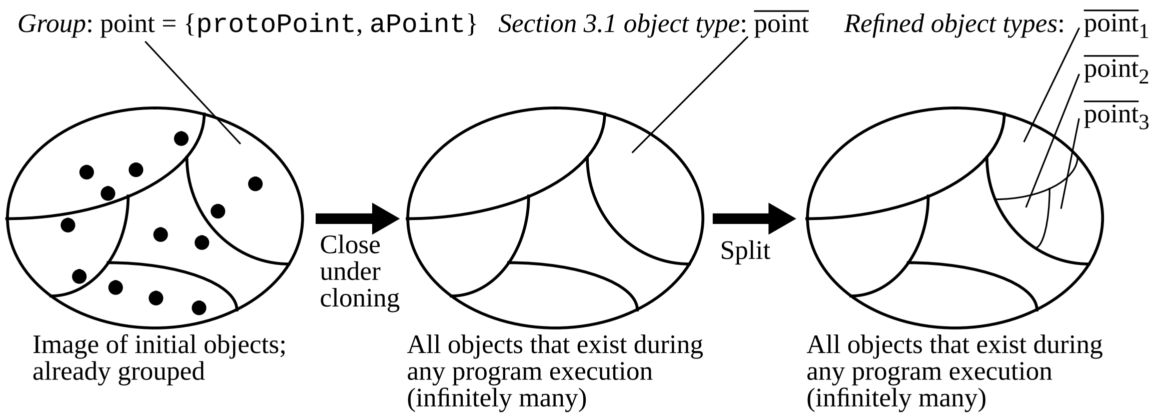

Group: point = {`protoPoint`, `aPoint`}
Section 3.1 object type: $\overline{\text{point}}$
Refined object types: $\overline{\text{point}_1}$ $\overline{\text{point}_2}$ $\overline{\text{point}_3}$
Close under cloning
Split
Image of initial objects; already grouped
All objects that exist during any program execution (infinitely many)
All objects that exist during any program execution (infinitely many)

Figure 39. From groups to object types to refined object types

In Section 3.1, objects were partitioned into groups that were closed under cloning to obtain object types (corresponding to the first arrow above). Here, object types are subsequently split into refined object types (corresponding to the second arrow).

After splitting, the point group gives rise to several refined object types. To make refined object types useful for analysis, any given object that exists during an execution of the target program should belong to ("have") exactly one refined object type. This uniqueness requirement can be rephrased in terms of two separate conditions:

- *Coverage*. Every object must belong to some refined object type.
- *Disjointness*. No object can belong to more than one refined object type.

Section 4.3.3 suggests that a relaxed disjointness condition might lead to more precise type inference.

We can now, finally, state precisely how splitting can improve precision of analysis of code with data polymorphism. By disjointness, assignments to the `x` coordinate in, say, $\overline{\text{point}_2}$ cannot affect the type of `x` in $\overline{\text{point}_3}$. Thus, if the type-inference algorithm carefully splits $\overline{\text{point}}$, it may be able to avoid mixing up integer and float points, and therefore infer more precise types for the methods that operate on only one kind of points. For example, the type-inference algorithm may be able to arrange that one of the refined object types, $\overline{\text{point}_1}$, say, represents point objects with integer coordinates, that $\overline{\text{point}}_2$ represents point objects with float coordinates, and finally that $\overline{\text{point}}_3$ represents point objects that may have either kind of coordinates.

This specific split represents the best-case scenario. Our framework also permits less "fortunate" splitting. For example, $\overline{\text{point}}$ could be split into three (or any number) of refined object types, all of which contain a mixture of integer and float points. The framework also permits unnecessary splitting. Thus, if a target program uses points with only integer coordinates, an overly eager algorithm for analyzing code with data polymorphism may still split $\overline{\text{point}}$ into several refined object types, all of which will have integer coordinates. Unnecessary splitting does not cause precision loss, but introduces redundancy and therefore slows down type inference.

Splitting is not simply the inverse of grouping, as the following example illustrates. Consider again the group:

$$\text{point} = \{\texttt{protoPoint}, \texttt{aPoint}\}.$$

A program that uses both integer and float points may create these by cloning either `protoPoint` or `aPoint` and subsequently assigning to the slots containing the coordinates. Thus, a finer *grouping* like

$$\begin{aligned}\text{protoPoint} &= \{\texttt{protoPoint}\}\\ \text{aPoint} &= \{\texttt{aPoint}\}\end{aligned}$$

would not (necessarily) help the type inferencer maintain separation of integer and float points. Indeed, in this example, the names of the two initial point objects suggest that the program may clone all its point objects, be they float- or integer points, from `protoPoint`. Nevertheless, for a target program that clones all its float points from `protoPoint` and all its integer points from `aPoint` or vice versa, the finer grouping would actually help the type inferencer maintain separation between integer and float points. In general, however, partitioning initial objects into finer groups cannot replace splitting.

Concept	Example		General	
	Section 3.1	**Now**	**Section 3.1**	**Now**
initial objects[a]	IM = {`nil`, `true`, `protoPoint`, `aPoint`, `47`, …}		$IM = \{\omega_1, \omega_2, \ldots, \omega_n\}$	
group[b]	point = {`protoPoint`, `aPoint`}		$\varnothing \subset G_i \subseteq IM$	
(refined) object types[c]	$\overline{\text{point}}$	$\overline{\text{point}}_1, \overline{\text{point}}_2, \overline{\text{point}}_3$	$\overline{G}_i$	$\overline{G}_{i1}, \overline{G}_{i2}, \ldots, \overline{G}_{ik_i}$
coverage[d]	$\overline{\text{point}} = \overline{\texttt{protoPoint}} \cup \overline{\texttt{aPoint}}$	$\overline{\text{point}}_1 \cup \overline{\text{point}}_2 \cup \overline{\text{point}}_3 = \overline{\texttt{protoPoint}} \cup \overline{\texttt{aPoint}}$	$\overline{G}_i = \bigcup_{\omega \in G_i} \overline{\omega}$	$\bigcup_{j=1}^{k_i} \overline{G}_{ij} = \bigcup_{\omega \in G_i} \overline{\omega}$
disjointness[e]	trivial	$\forall j \neq j' : \overline{\text{point}}_j \cap \overline{\text{point}}_{j'} = \varnothing$	trivial	$\forall j \neq j' : \overline{G}_{ij} \cap \overline{G}_{ij'} = \varnothing$

Table 14. Splitting of object types

a. From Section 3.1.2: initial objects are the objects in the image, *IM*, created prior to execution of the target program.
b. From Section 3.1.2: groups *partition* the initial objects according to similarity. Thus, groups are non-empty, pair-wise disjoint, and cover the image.
c. In Section 3.1.2, one object type was obtained from each group by closure under cloning; to analyze code with data polymorphism we subsequently split each closed groups into a finite number of refined object types.
d. In words: every object has an object type.
e. In words: no object has more than one object type.

Our framework for analyzing code with data polymorphism is summarized in Table 14. The table develops the framework both in general and for the point example that was given above. In addition, the table contrasts the new framework (the columns labelled "Now") with the original framework (the columns labelled "Section 3.1").

Commonly, the splitting of object types is performed at the granularity of *allocation points.* Thus, two objects have the same refined object type if:

- They are cloned, perhaps indirectly, from members of the same group; and
- They are allocated at the same point in the program (for specificity, we consider all initial objects as being allocated at the same program point). This requirement can be strengthened to produce a finer partitioning by applying program expansions or comparing call strings leading to the allocation points.

The first requirement formed the complete definition of object types in Section 3.1. The second requirement now refines object types. Returning to the point example, if the target program clones point objects in two different places, the basic type-inference algorithm assigns them all the same object type, $\overline{\text{point}}$, whereas an algorithm that uses allocation points to target data polymorphism assigns different object types to the objects created at these two places.

Allocation points are useful because they guarantee both coverage (objects are either initial or allocated somewhere) and disjointness (an object cannot be allocated at more than one point). They have been used in other analyses than constraint-based type inference [29, 31, 129]. However, our framework does not *require* that splitting is based on allocation points. It permits any splitting approach that assigns each object a unique object type from a finite set of object types.

When should objects be mapped to their refined object type? If splitting is based on allocation points, objects can be mapped immediately after their creation. For example, when several different allocation points in a program may clone a member of the (unrefined) $\overline{\text{point}}$ object type (the object `protoPoint`, say), the resulting objects can be assigned different refined object types, as follows:

type(`protoPoint _Clone`) = $\{\overline{\text{point}_1}\}$ at allocation point #1,
type(`protoPoint _Clone`) = $\{\overline{\text{point}_2}\}$ at allocation point #2, and
type(`protoPoint _Clone`) = $\{\overline{\text{point}_3}\}$ at allocation point #3.

If splitting is based on a different property than allocation points, the type-inference algorithm must deal with objects that change object type. Upon cloning, new objects are initially mapped to the unrefined type. After flowing sufficiently far to be discerned, their object type must be switched to the appropriate refined object type. This switching of object types must be handled carefully to avoid conflicts with the disjointness criteria outlined above and should preferably happen before any assignments "pollute" the types of the instance variables in the unrefined object type. (Note: object types are switched from an unrefined to a refined object type so switching does not violate the disjointness requirement of refined object types: no object ever has more than one refined object type. However, see Section 4.3.3 for a brief discussion of a more general class of type-inference algorithms that allow objects to change refined object types.)

4.3.2 Algorithms for analyzing code with data polymorphism

With the extended framework in place, the existing algorithms for analyzing code with data polymorphism can be characterized concisely. These algorithms were also reviewed as related work in Section 2.3.2; the following summary is meant to illustrate the use of the framework, rather than exhaustively survey the field of related work.

Oxhøj, Palsberg, and Schwartzbach proposed a 1-level class expansion algorithm [85]: two objects allocated at different program points in the original (unexpanded) program always have different object types. Thus, if the program contains m allocation points and n classes, their algorithm operates with mn different object types. Much like the 1-level expansion algorithm for parametric polymorphism, the 1-level class expansion has serious deficiencies:

- *Imprecision.* It fails to resolve data polymorphism when the objects containing the polymorphic variables are allocated at the same program point.
- *Inefficiency.* It may unnecessarily split object types, resulting in redundant analysis and inefficiency. For example, most places that allocate point objects probably operate with integer points; thus, they should be assigned the same object type to ensure efficiency. To reduce but not entirely eliminate the redundancy, it has been suggested that the

programmer should annotate certain classes as “collections” (read: “having polymorphic instance variables”). The algorithm then need only perform the 1-level class expansion for these classes [84].

Phillips and Shepard generalized the class expansion from a single level to *k* levels, thereby improving the precision, but also increasing redundancy and cost.

Plevyak and Chien proposed a more powerful iterative algorithm for analyzing code with data polymorphism [93]. This data iterative algorithm avoids redundancy by splitting an object type only if it can determine, based on a previous iteration, that a precision gain will be achieved. For example, if a program allocates point objects at several program points, but in all cases uses these with integer coordinates, the data iterative algorithm can recognize this fact and avoid splitting the $\overline{\text{point}}$ object type unnecessarily. Moreover, when an object type must be split, it can be done with greater precision than splitting it *m*-way where *m* is the number of allocation points: if there are only two different ways that point objects are used, the data iterative algorithm may be able to determine that $\overline{\text{point}}$ should be split 2-way.

We defined adaptive algorithms as algorithms that employ partial type information to decide when to create additional templates (Section 3.2.5). This definition can be extended to data polymorphism: a *data-adaptive algorithm* is one that employs partial type information to decide how to split object types. In this light, the data-iterative algorithm is data-adaptive, whereas the 1-level class expansion is not.

Although the cartesian product algorithm itself does not address data polymorphism, neither does it conflict with the existing algorithms for analyzing data polymorphism. Hence, one could combine CPA with any of these algorithms to obtain a comprehensive type-inference system. A particularly interesting question to investigate is how the (iterative, data iterative) combination compares with the (cartesian product, data iterative) combination. The answer is not obvious. The former combination may work well in practice, because the two kinds of iterations can develop the constraint network in parallel. Or the latter combination may excel because CPA’s strong handling of parametric polymorphism clears away the brush, giving the data iterative part immediate access to dig out its potatoes. It is still an open issue whether a CPA-like data-adaptive algorithm, or indeed any non-iterative data-adaptive algorithm, can be developed.

4.3.3 Discussion

Data polymorphism refers to the ability of slots to hold objects of different types. Our framework covers type-inference algorithms that split object types to analyze data polymorphism more precisely than the basic algorithm. Splitting can gain precision by replacing a single type for a polymorphic slot with several smaller types. The finer the object type containing the polymorphic slot is split, the better the precision that may be achieved. Splitting indiscriminatingly, however, can result in slow type inference.

Even when object types are split, the members of a refined object type may still exhibit data polymorphism. Some data structures are truly polymorphic, and no amount of splitting can change this fact. For example, if a program uses a slot in a *single* object to sometimes hold smallInts and sometimes floats, splitting cannot prevent accesses to this slot from seeing both $\overline{\text{smallInt}}$ and $\overline{\text{float}}$ as possible results. However, flow-sensitive analysis, the topic of Section 4.5, may improve precision in this case by allowing slots to have different types at different points in the target program. More specifically, flow-sensitive analysis tracks the flow of control from assignments to reads of slots, attempting to eliminate certain object types from the types of slots when control-flow properties make it impossible for them to reach specific reads.

A flow-sensitive algorithm assigns different types to the same slot at different points in the program. It does not, however, assign different object types to the same object at different points in the program. An object has the same object type from birth to death—no matter how dramatically the types of its slot are changed by assignments through the lifetime of the object. In contrast, an analysis algorithm that can change the (refined) object type of objects whose slots are assigned would seem to be able to maintain tighter types of slots. Upon encountering an assignment to a slot of an object α, the algorithm would have two options: extend the type of the slot in the current object type of α or switch the object type of α to avoid polluting the slot’s type in the current object type. To some extent, the capabilities of this algorithm would subsume both the splitting of object types and the flow-sensitive analysis. This algorithm would require the disjointness property of our framework to be relaxed. Instead of demanding that each object has at most one object type throughout the program, the relaxed disjointness condition could state that each object has at most one object type at each program point. However, developing such an algorithm is future work.

4.4 Blocks

Blocks play an important role in all but the simplest Self programs; they are used to implement conditional statements, case statements, loops, and exception handling. Consequently, blocks occur in large numbers and contain a significant fraction of all expressions. Table 15 quantifies this claim for our benchmark Self programs. For each program, the table gives the number of expressions analyzed during type inference, the number of blocks encountered, and the percentage of the expressions that are in blocks. Each expression in the program is counted only once even if the method containing it was analyzed several times. The table shows that close to half of all expressions analyzed are found in blocks.

	#expressions[a]	**#blocks**	**%exp's in blocks**
HelloWorld	23	2	13
Factorial	3,870	280	48
Richards	4,720	329	46
DeltaBlue	4,901	343	48
Diff	7,317	531	47
PrimMaker	7,887	502	39

Table 15. The frequency of expressions in blocks and the number of blocks

a. This count includes the blocks (blocks are expressions in Self).

Blocks have several distinguishing properties that affect type inference, including:

- *Two-phase evaluation.* Like closures in Scheme, blocks evaluate in two phases. Phase one occurs when the block expression `[bodyOfBlock...]` is encountered. The result is a *closure,* i.e., a pair (*b*, *l*) where *b* is the block and *l* is a *lexical pointer* to the activation record in which the block was closed. The expressions between the brackets (a.k.a. the *block method*) are not executed. The closure can be passed to methods and stored in slots like any other object. Phase two occurs later, when the closure receives the `value` message (or `value:` or `value:With:` etc., depending on how many arguments the block needs). Now the block method `bodyOfBlock...` executes. Phase two may be repeated arbitrarily often as this expression illustrates:

  ```
  10 do: [|:i| i factorial printLine].
  ```

- *Lexical access.* Expressions in blocks may read and write slots defined in their lexical environment. The slots are accessed via the lexical pointer of the invoked closure.

- *Non-local return.* Blocks may perform non-local returns. Upon completing phase two of a block with a non-local return, control (and the result) does not return to the `value` send that invoked the block method. Instead, it returns to the send that invoked the *outer* method containing the block. Non-local returns are used extensively in Self programs, including to terminate loops and handle exceptions.

Type inference for blocks and expressions in blocks must respect these properties. Section 4.4.1 explains our technique for ensuring precision of type inference of code with blocks. Section 4.4.2 addresses efficiency of type inference. A further concern is that of ensuring *termination* when analyzing recursive methods that use blocks in certain ways. We defer this issue to Chapter 5.

4.4.1 Obtaining precision

Consider the following "safe" `inverse` method, which returns the inverse of the receiver, or the receiver when it is zero:

```
inverse = (
  0 = self ifTrue: [^ self].
  1 / self.
).
```

Invoking `inverse` on a smallInt always results in a smallInt (the only possible values happen to be 0 and ±1, but this is less important). Invoking `inverse` on a float always produces a float, even in the zero case, because the expression determining the returned value is `self`.

The `inverse` method illustrates how to analyze blocks precisely. The method constitutes a minimal example, involving one block, `[^ self]`, one lexical access (the expression `self` in the block accesses the slot `self` in the lexically enclosing `inverse` method), and one non-local return. Although in this particular example, the lexically accessed slot is the receiver `self`, there is nothing special about this slot with respect to the following discussion. Had `inverse` been rewritten (as `inverse:`) to ignore the receiver and instead invert a non-receiver argument, the following discussion would remain the same. The block in `inverse` does not contain any assignments to slots defined in its lexical environment. However, the techniques developed to handle read accesses to slots in the lexical environment of blocks carry over straightforwardly to assignments by simply "reversing the constraint arrows."

We will need to distinguish between slots and expressions that access these slots. However, both concepts have the same syntax in Self. To clarify the description, we augment the syntax and let $\texttt{self}_{\text{slot}}$ denote the slot `self` and $\texttt{self}_{\text{exp}}$ an expression accessing this slot. The augmented syntax plays no role in the analysis of `inverse`; it is merely a notational convenience. With this more specific notation, and explicitly showing the receiver formal parameter, the `inverse` method becomes:

```
self$_{slot}$ inverse = (
  0 = self$_{exp}$ ifTrue: [^ self$_{exp}$].
  1 / self$_{exp}$.
).
```

Assume now that the type inferencer has created two templates, *S* and *T*, for `inverse`. *S* describes the computation of the inverse of an integer, and *T* describes the computation of the inverse of a float. Thus, in *S* the type of $\texttt{self}_{\text{slot}}$ is $\{\overline{\text{smallInt}}\}$ and in *T* it is $\{\overline{\text{float}}\}$[†]. We express these types by subscripting with the template that provides the context, i.e.,

$$\text{type}_S(\texttt{self}_{\text{slot}}) \;=\; \{\overline{\text{smallInt}}\}$$
$$\text{type}_T(\texttt{self}_{\text{slot}}) \;=\; \{\overline{\text{float}}\}\,.$$

Given *S*, *T*, and a template for the block method of $\texttt{[^ self}_{\text{exp}}\texttt{]}$, what should the type of the lexical access $\texttt{self}_{\text{exp}}$ be? The simplest approach, illustrated in Figure 40, assumes that $\texttt{self}_{\text{exp}}$ may access $\texttt{self}_{\text{slot}}$ in *any* `inverse` template—i.e., in *S* and *T* (and any future `inverse` templates). Under this assumption, the best type that can be inferred for $\texttt{self}_{\text{exp}}$ is $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$, since:

$$\text{type}(\texttt{self}_{\text{exp}}) \supseteq \text{type}_S(\texttt{self}_{\text{slot}}) \cup \text{type}_T(\texttt{self}_{\text{slot}}) = \{\overline{\text{smallInt}}, \overline{\text{float}}\}.$$

Before deciding whether this type is sufficiently precise, consider how non-local returns are handled in Figure 40. Recall that a non-local return does not return to the send that invoked the block method. Instead it returns to the send that invoked the regular method (i.e., non-block method) that lexically encloses the block with the non-local return. The constraints in the figure capture the semantics of non-local returns with a constraint from the result type of the block with the non-local return to the result type of the enclosing regular method. Analogous to the assumption that lexical accesses may access any of the templates for the enclosing method, the non-local return constraints were generated under the conservative assumption that the value returned non-locally by $\texttt{[^ self}_{\text{exp}}\texttt{]}$ can become the value of *any* `inverse` template (the figure includes non-local return constraints to both *S* and *T*).

[†] Recall from Section 3.2.1 that the type-inference algorithm computes types for both expressions and slots.

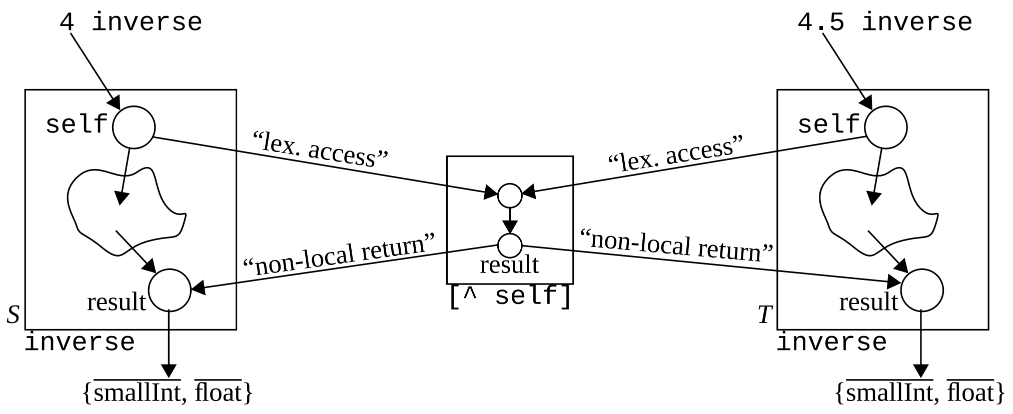


Figure 40. Pessimistic type inference of lexical accesses

Precise type inference is not possible under the assumption that lexically accesses may access the variable in any template of the enclosing method containing the slot.

The constraints in Figure 40 resulted from two conservative assumptions: that lexical accesses may access the slots in any template for the relevant lexically enclosing method, and that non-local returns may go to any template for the relevant lexically enclosing regular method. Unfortunately, precise types cannot be inferred under these conservative assumptions. We saw above that the type inferred for `self`$_{\text{exp}}$ in the block method template is (a superset of) {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}. Consequently, since this type is returned non-locally to both of the `inverse` templates, they both have result types {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}. Clearly, the precise types should have been {$\overline{\text{smallInt}}$} for the *S* template (since the inverse of a smallInt is a smallInt) and {$\overline{\text{float}}$} for the *T* template (since the inverse of a float is a float). Instead, the conservative assumptions caused the inferencer to mix up the two cases and lose precision for both of them. In Self programs, non-local accesses and non-local returns occur too frequently to permit this level of inaccuracy.

To improve precision when analyzing lexical accesses and non-local returns, the type inferencer must emulate the two-phase evaluation of blocks. In particular, it must employ lexical pointers similar to what happens during execution:

- *Phase one*. During execution, blocks evaluate to closures (*b*, *l*). Thus, we define a type-inference-time version of closures, *closure object types*: (*b*, *L*). The lexical pointer *L* is critical: instead of referring to an activation record (which only exists at run time), it refers to a template. Recall from Table 6 that templates correspond to activation records. For example, the type of the expression `[^ self`$_{\text{exp}}$`]` in the `inverse` template *S* is the set consisting only of the closure object type (`[^ self`$_{\text{exp}}$`]`, *S*), i.e.:

  $$\text{type}_S(\texttt{[\^{} self}_{\text{exp}}\texttt{]}) = \{\ (\texttt{[\^{} self}_{\text{exp}}\texttt{]}, S)\ \}.$$

  Similarly, in the template *T* the type of the same block is:

  $$\text{type}_T(\texttt{[\^{} self}_{\text{exp}}\texttt{]}) = \{\ (\texttt{[\^{} self}_{\text{exp}}\texttt{]}, T)\ \}.$$

- *Phase two*. Assume the type inferencer has encountered a send expression with the selector `value` that has a closure object type as the receiver. In other words, the type inferencer is about to analyze a block method invocation. The type inferencer creates a template for the block method and copies the lexical pointer from the receiver closure object type into the template; see Figure 41. When the inferencer later analyzes the block method template, the lexical pointer supports precise type inference of lexical accesses because it specifies the templates in which the accessed slots are found. For example, in a template for the closure object type (`[^ self`$_{\text{exp}}$`]`, *S*), the lexical pointer specifies that the expression `self`$_{\text{exp}}$ accesses `self`$_{\text{slot}}$ in the *S* template. Consequently:

  $$\text{type}(\texttt{self}_{\text{exp}}) = \text{type}_S(\texttt{self}_{\text{slot}}) = \{\overline{\text{smallInt}}\}.$$

Compare this type with the less precise {$\overline{\text{smallInt}}$, $\overline{\text{float}}$} that was inferred for type(`self`$_{\text{exp}}$) in Figure 40 under the pessimistic assumption that `self`$_{\text{exp}}$ could access any `inverse` template.

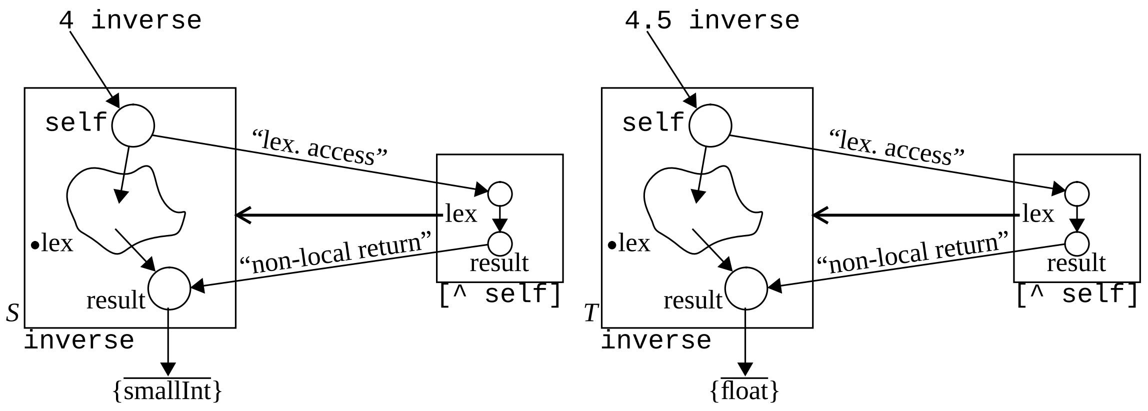


Figure 41. Precise type inference of lexical accesses

The type inferencer simulates the use of lexical pointers to ensure precision of type inference. The lexical pointers precisely guide lexical accesses and non-local returns.

The `inverse` method contains no blocks nested more than one level deep. In general, however, blocks may be nested inside blocks to an arbitrary depth. Our approach generalizes straightforwardly to handle deeply nested blocks. The type inferencer traverses the linear chain of lexical pointers from template to template until it arrives at the one that contains the accessed slot or the one that the non-local return constraint should go to.

Another complication, also absent from the simple example in Figure 41, is the possibility of multiple block method templates for a single closure object type. When analyzing polymorphic sends, the type inferencer always generates cartesian products to maximize precision. Thus, if a polymorphic send invokes a block closure, the type inferencer generates multiple templates from the closure object type that appears as the receiver of the send. All of these templates have the same lexical pointer, since it is determined by the lexical pointer in receiver closure object type. The lexical pointers connect the templates into a *forest*; see Figure 42. Templates for regular methods, which have no lexically enclosing methods, root the trees. A template at depth *i* in a lexical tree corresponds to a block nested *i* levels deep syntactically.

In summary, we have found that maintaining a close parallel with the execution model, as we do for activation records and templates, permits precise analysis of blocks. Table 16 summarizes the correspondence between execution and type inference. It supplements Table 6 on page 45.

	Run time	**Type-inference time**
Result of `[block-body]` is a	closure	closure object type
Sending value to a closure yields	block method activation record	block method template
Lexical pointer refers to	activation record	template
Non-local access follows	lexical pointers to proper activation record	lexical pointers to proper template
Non-local return	becomes result of outer method invocation	propagates type into outer method template's result type

Table 16. How type inference emulates execution to precisely analyze blocks

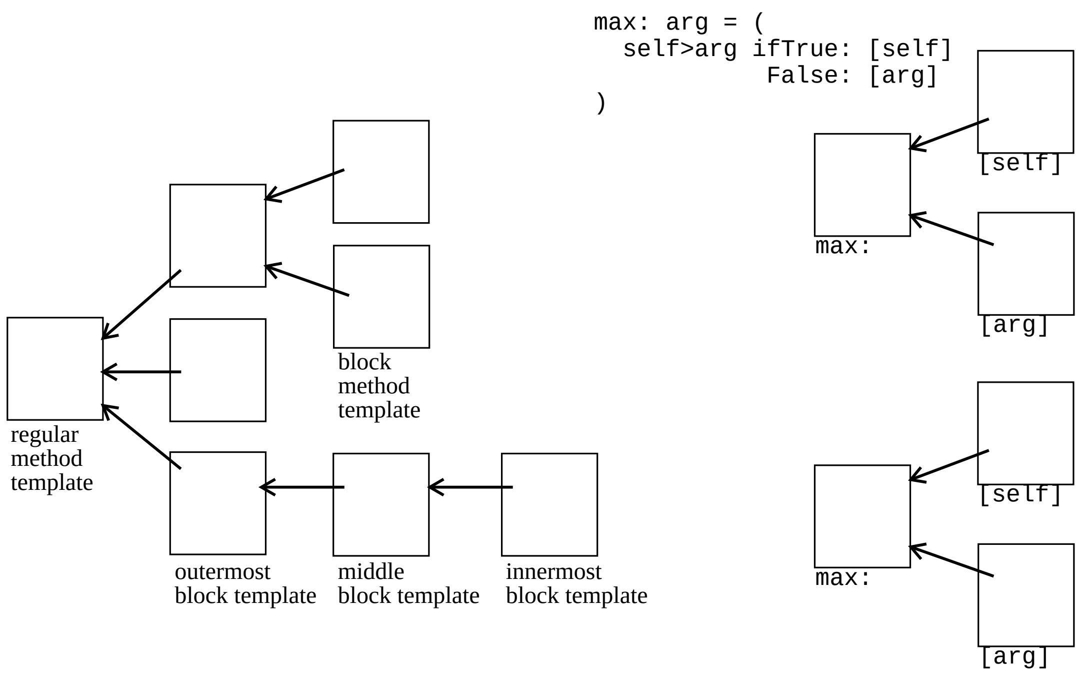


Figure 42. Tree structure of lexical pointers in templates

The lexical pointers arrange the templates into trees. Each tree is rooted by a regular (i.e., non-block) method template. The trees' structure match the nesting of blocks, as illustrated by the `max:` method.

4.4.2 Improving efficiency

The efficiency of type inference depends on the number of closure object types created. There are two reasons:

- A large fraction of all expressions are found in blocks. Hence, creating many closure object types means that many expressions may need to be analyzed many times.
- Adaptive type-inference algorithms suffer from "positive feedback." Creating more closure object types increases the need for templates (since templates are customized to specific types of arguments). In turn, more templates cause the creation of more closure object types. In the worst case, the feedback may interact with recursion and lead to non-termination, a problem that will be addressed in Chapter 5.

Consequently, to maintain efficiency, the type inferencer should create no more closure object types than are necessary. There are two extremes:

- One closure object type per block; this closure object type must then be general enough to describe all possible cases during execution. In Section 4.4.1, this approach was rejected as being insufficiently precise.
- A distinct closure object type for each template of a method that contains a block. This approach is the precise alternative suggested in Section 4.4.1.

We have found that the second approach is almost right, but can be tuned for higher efficiency without impacting precision. The idea is to reduce the number of closure object types by *fusing* them when this cannot lead to loss of precision. The `max:` method will illustrate the idea (again annotated with "exp" and "slot" for clarity):

$\texttt{self}_{\text{slot}}\ \texttt{max: arg}_{\text{slot}}\ \texttt{= (self}_{\text{exp}}\texttt{>arg}_{\text{exp}}\ \texttt{ifTrue: [self}_{\text{exp}}\texttt{] False: [arg}_{\text{exp}}\texttt{]).}$

Assume we have four templates for `max:`, *S*, *T*, *U*, and *V*, with the types shown in Table 17.

Template	**Type of $\texttt{self}_{\text{slot}}$**	**Type of $\texttt{arg}_{\text{slot}}$**
S	$\{\overline{\text{smallInt}}\}$	$\{\overline{\text{smallInt}}\}$
T	$\{\overline{\text{float}}\}$	$\{\overline{\text{float}}\}$
U	$\{\overline{\text{smallInt}}\}$	$\{\overline{\text{float}}\}$
V	$\{\overline{\text{float}}\}$	$\{\overline{\text{smallInt}}\}$

Table 17. Four `max:` templates with different types of receivers and arguments

A priori, since we have four `max:` templates, there should be four closure object types for each of the block expressions $[\texttt{self}_{\text{exp}}]$ and $[\texttt{arg}_{\text{exp}}]$:

$([\texttt{self}_{\text{exp}}], S)$ $([\texttt{self}_{\text{exp}}], T)$ $([\texttt{self}_{\text{exp}}], U)$ $([\texttt{self}_{\text{exp}}], V)$
$([\texttt{arg}_{\text{exp}}], S)$ $([\texttt{arg}_{\text{exp}}], T)$ $([\texttt{arg}_{\text{exp}}], U)$ $([\texttt{arg}_{\text{exp}}], V)$

However, it is easy to see that two closure object types for each block would suffice. The types in the $[\texttt{self}_{\text{exp}}]$ block depend only on the type of $\texttt{self}_{\text{slot}}$, which is the same in *S* and *U*, and in *T* and *V*. Acknowledging this, we fuse $([\texttt{self}_{\text{exp}}], S)$ and $([\texttt{self}_{\text{exp}}], U)$ to obtain $([\texttt{self}_{\text{exp}}], \{S, U\})$. Likewise, $([\texttt{self}_{\text{exp}}], T)$ and $([\texttt{self}_{\text{exp}}], V)$ can be fused to $([\texttt{self}_{\text{exp}}], \{T, V\})$. After fusing, the complete set of closure object types is:

$([\texttt{self}_{\text{exp}}], \{S, U\})$ $([\texttt{self}_{\text{exp}}], \{T, V\})$
$([\texttt{arg}_{\text{exp}}], \{S, V\})$ $([\texttt{arg}_{\text{exp}}], \{T, U\})$

Fusing closure object types saves analysis effort because there are fewer templates for the involved blocks to analyze. Moreover, fusing lessens the need for other templates; in the `max:` example, fusing eliminates several `ifTrue:False:` templates.

To determine which closure object types can be fused, the types of the slots accessed in the lexical environment must be known (to avoid fusing closure object types that access differently-typed slots in their lexical environment). However, these types are still being computed when the fusing must be decided. It is straightforward to apply iteration to overcome this difficulty:

- the first iteration fuses all closure object types for a given block, i.e., is equivalent to the imprecise analysis of blocks;
- subsequent iterations fuse less, using the previous iteration's types to guide the fusing.

However, the iteration overhead may more than offset the speedup from the fusing, so in the Self type inferencer we took a different approach. The cartesian product algorithm creates templates with monomorphic argument types. Consequently, the types of argument slots are known immediately, and closure object types that only access argument slots (and constant slots) in their lexical environment, such as the two in the `max:` method, can be fused non-iteratively. For lexical accesses to non-argument slots we conservatively assume that they may be differently typed. Thus, the type inferencer avoids iteration, but sometimes misses an opportunity to fuse closure object types.

Fusing complicates the conceptual model of type inference slightly, since closure object types now have multiple lexical pointers. Therefore, the lexical pointers no longer arrange templates into a collection of trees. Instead, they form a collection of acyclic directed rooted graphs; see Figure 43. For Self we found the performance gain of fusing high enough to justify the added complexity: Table 18 shows that for our benchmark programs fusing speeds up type inference by a factor of approximately 1.4.

Although fusing does not cause precision loss of the inferred types, it *does* introduce a more subtle information loss. After fusing, less is known about the types of the slots in the lexical environment that the fused closure object types *ignore*. For example, in a template for a fused closure such as $([\texttt{arg}_{\text{exp}}], \{S, V\})$, it is impossible to tell whether the type of $\texttt{self}_{\text{slot}}$ in the lexically enclosing method is $\{\overline{\text{smallInt}}\}$ (the *S* case) or $\{\overline{\text{float}}\}$ (the *V* case). In contrast, without fusing, there would have been two separate closures $([\texttt{arg}_{\text{exp}}], S)$ and $([\texttt{arg}_{\text{exp}}], V)$ and therefore two separate templates for the block method, in each of which the type of $\texttt{self}_{\text{slot}}$ would be known exactly. Is it ever important to know the types of ignored slots in the lexical environment of fused closure object types? Occasionally. If

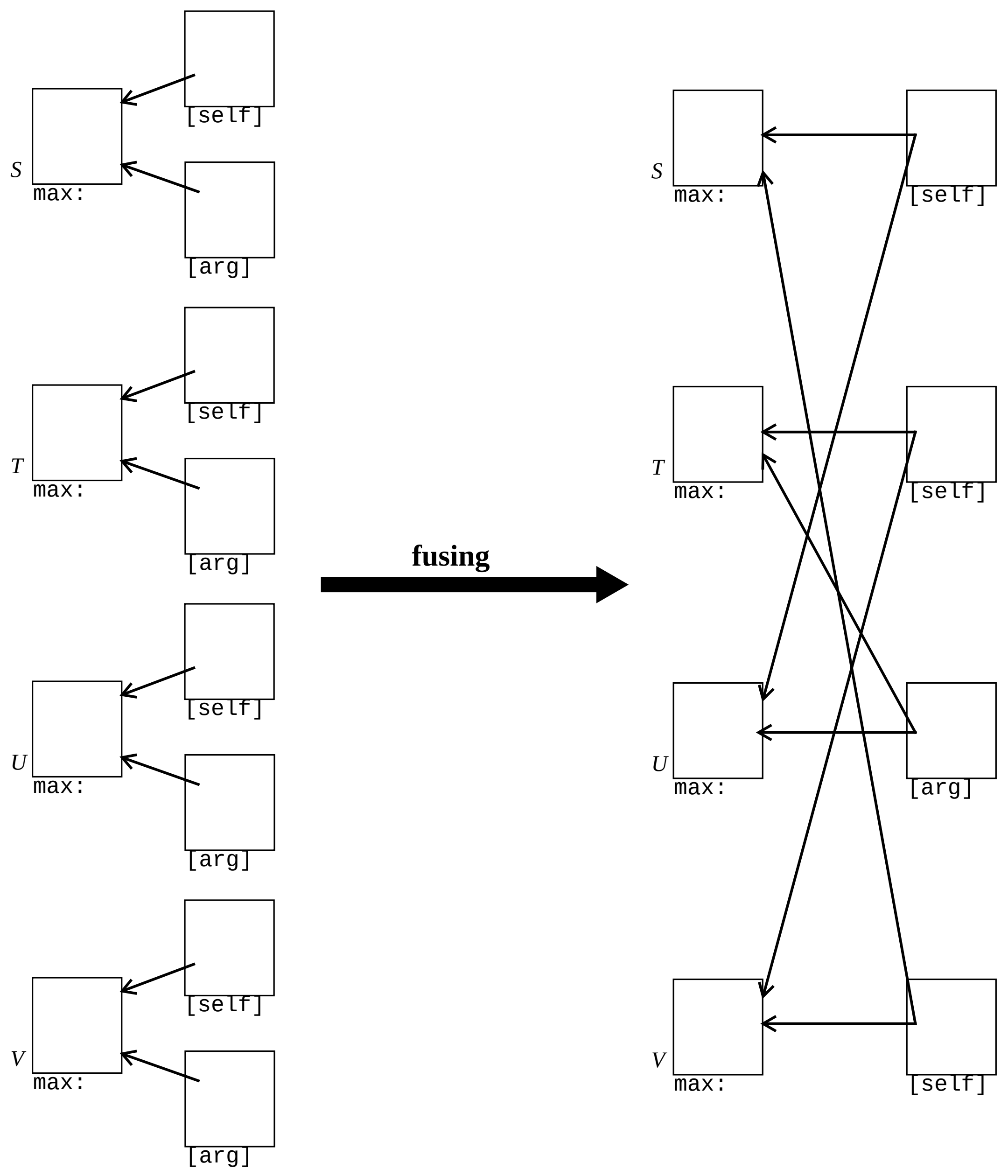


Figure 43. Before and after fusing closure object types

After fusing closure object types, there are fewer of them, and therefore fewer block method templates. This reduction speeds up type inference, but also complicates the structures slightly. The lexical pointers that used to connect templates into trees now connect them into directed acyclic graphs.

type inference guides compilation, in languages with non-uniform sizes of object references, even an ignored slot could have an indirect effect, since its contents may influence memory layout.

In conclusion, while fusing speeds up type inference by about 30%, it complicates the conceptual model somewhat, and incurs a subtle loss of information.

	CPU seconds measured on a 167 MHz UltraSPARC		**Speedup factor**
	Type inf. time without fusing	**Type inf. time with fusing**	
HelloWorld	0.0	0.0	n/a
Factorial	9.5	6.8	1.4
Richards	10.3	7.2	1.4
DeltaBlue	15.1	10.4	1.5
Diff	22.2	15.5	1.4
PrimMaker	13.3	9.6	1.4

Table 18. The effect on type inference performance of fusing closure object types

4.5 Flow-sensitive analysis of variable[†] accesses

Assignments are “non-destructive” in type inference. They do not erase the old type of the assigned variable, so object types keep accumulating in the types of variables[‡]. For example, if a variable first holds booleans and later smallInts, it has type {$\overline{\text{false}}$, $\overline{\text{true}}$, $\overline{\text{smallInt}}$}. Clearly, even if the variable may hold both integers and booleans, it is possible that some reads of the variable retrieve integers only and others retrieve booleans only. Based on this example, we distinguish between two different type-inference approaches:

- *Assigning types to variables.* In this flow-insensitive approach, variable accesses (reads) always have the same type as the variable they access. In the example given above, all reads of the variable have type {$\overline{\text{false}}$, $\overline{\text{true}}$, $\overline{\text{smallInt}}$}.
- *Assigning types to variable accesses.* In this flow-sensitive approach, variable accesses need not have the full type of the accessed variable. The type of a particular variable access can be sharpened if the inferencer (somehow) can determine that the access can only return a subset of the kinds of objects that the variable may hold throughout the program. In the example outlined above, the reads that always retrieve integers can have the more precise type {$\overline{\text{smallInt}}$}, and the reads that retrieve booleans can have the type {$\overline{\text{false}}$, $\overline{\text{true}}$}.

The same distinction was made in the review of related work (Section 2.3.3). There it was argued that the extra power of the second alternative can be important when adding a type system to an existing body of code. Four increasingly powerful approaches to typing variable accesses are described below: flow-insensitive (Section 4.5.1), eliminate dead initializers (Section 4.5.2), use static single assignment (SSA) form (Section 4.5.2), and use reaching definitions (Section 4.5.4).

We have implemented the first three approaches, but only for local slots. (Instance slots are harder to deal with because they may represent multiple locations; however, it may be possible to generalize Chase et al’s techniques for introducing strong updates [29].) Our results indicate that the simple elimination of dead initializers pays off significantly, and that SSA is slightly better still. Table 19 presents data obtained on the Self system. The table gives the number of templates that different variants of the cartesian product algorithm generate when analyzing a set of test programs. The number of templates directly reflects precision of type inference as follows: less precise types have more members, making the cartesian products larger, and therefore causing the cartesian product algorithm to create more templates. Since the flow-insensitive algorithm creates the most templates, it is the least precise. For the non-trivial test programs, turning on dead initializer elimination reduces the number of templates by 15-25%; turning on SSA transformation reduces the number of templates only slightly more.

[†] In this section we use the common word *variable* to denote an assignable slot.

[‡] Chase, Wegman, and Zadeck [29] distinguish between *strong* and *weak updates*. Strong updates replace the old type with the type of the assigned value, whereas weak updates merge the two types. Chase, Wegman, and Zadeck also give conditions under which an analyzer can use strong updates to gain precision.

	Flow-insensitive	Dead initializer elim.	SSA transformation
HelloWorld	7 / 0.0%	7 / 0.0%	7 / 0.0%
Factorial	7,377 / 0.0%	6,217 / 15.7%	6,208 / 15.8%
Richards	7,959 / 0.0%	6,726 / 15.5%	6,717 / 15.6%
DeltaBlue	11,954 / 0.0%	9,150 / 23.5%	9141 / 23.5%
Diff	17,685 / 0.0%	12,817 / 27.5%	12,660 / 28.4%
PrimMaker	10,195 / 0.0%	8,520 / 16.4%	8,511 / 16.5%

Table 19. Number of templates / percent saved over flow-insensitive analysis

4.5.1 Laissez faire (flow-insensitive analysis)

The last assignment to a variable determines the value retrieved when the variable is read. The simplest but also least precise analyzer ignores this sequencing of assignments and reads. It always assigns the full type of the accessed variable to reads. Figure 44 shows a method, `five:`, and the constraints relevant to determining the type of the variable access `x` (the last expression in `five:`)[†]. The type of the access is obtained from the type of the variable which in turn is determined by the types of the three expressions that may define its value: `x<-'five'`, `x: 5`, and `x: 5.0`. For uniformity, we treat the variable's initializer as an assignment rather than seeding the type variable for `x` with the initial value's object type; see Step 2 in Section 3.2.1.

```
five: arg = (
  |x<-'five'|
  arg ifTrue: [x: 5] False: [x: 5.0].
  x.
).
```

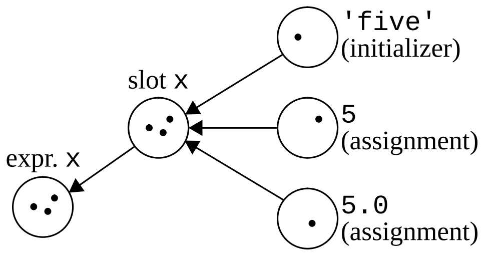


Figure 44. Flow-insensitive analysis of variable accesses

With flow-insensitive analysis, the type of a variable access is obtained from the type of the variable which, in turn, is the union of the types of all expressions that may define the value of the variable.

For typical Self programs, we found the flow-insensitive approach inadequate. For example, it infers the type {$\overline{\text{string}}$, $\overline{\text{smallInt}}$, $\overline{\text{float}}$} for the result of `five:`, and even a trivial method such as

```
four = ( |v| v: 4. v. )
```

cannot be analyzed precisely: the type of the last expression `v` is {$\overline{\text{nil}}$, $\overline{\text{smallInt}}$}. While the lack of precision could perhaps be tolerated locally, it cannot be isolated. Sends that invoke `four` also have $\overline{\text{nil}}$ in their type and so do expressions that consume the result of these sends.

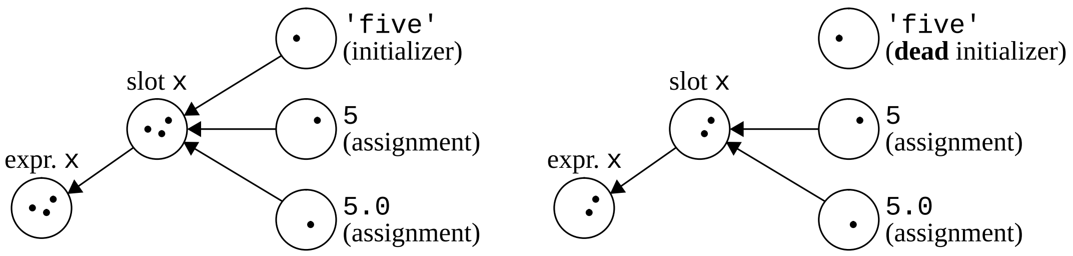
4.5.2 Eliminate dead initializers

A variable has a dead initializer when its initial value cannot be retrieved by any read. In other words, a variable's initializer is dead if the variable is guaranteed to be assigned to before being read. For example, the `v` slot in the `four` method above has a dead initializer (an implicit one, since `|v|` in Self is shorthand for `|v<-nil|`). Since

[†] The `five:` method may seem peculiar. It will be used as a running example not for its realism but because it is the minimal example that sheds light on how the approaches for flow-sensitive analysis of variable accesses differ.

dead initializers cannot affect the execution of the program, they can safely be omitted from type inference. Subsequently, variables have smaller types, in turn leading to more precise type inference for reads. For example, with the elimination of dead initializers, the type inferred for `four` is $\{\overline{\text{smallInt}}\}$ instead of $\{\overline{\text{nil}}, \overline{\text{smallInt}}\}$, since $\overline{\text{nil}}$ is never added to the type variable for `four`'s `v` slot. Figure 45 illustrates the effect of dead initializer elimination on the slightly more interesting `five:` method. After elimination of `x<-'five'` (which is a dead initializer because whether the argument is `true` or `false`, `x` is assigned before it is read), the type inferred for `five:` improves from $\{\overline{\text{string}}, \overline{\text{smallInt}}, \overline{\text{float}}\}$ to $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$.

```
five: arg = (
  |x<-'five'|
  arg ifTrue: [x: 5] False: [x: 5.0].
  x.
).
```



without elimination of dead initializers:
type(x_{expr}) = $\{\overline{\text{string}}, \overline{\text{smallInt}}, \overline{\text{float}}\}$

with elimination of dead initializers:
type(x_{expr}) = $\{\overline{\text{smallInt}}, \overline{\text{float}}\}$

Figure 45. Elimination of dead initializers

Elimination of dead initializers prevents them from "polluting" the types of expressions that access the variables (and other expressions downstream).

Dead initializers are eliminated in two steps:

- Find slots that have dead initializers.
- For each slot found, omit the constraint from the initializer expression's type to the slot's type.

Exact detection of dead initializers is impossible since liveness can depend on the outcome of general boolean expressions used in conditional statements. Instead, we must approximate conservatively. However, even approximating can be surprisingly hard and/or become surprisingly inaccurate. Consider again Figure 45. While it may seem safe to declare `x`'s initializer dead as we did for illustration purposes above, this reasoning is flawed! If `five:` is called with an argument that implements `ifTrue:False:` as a no-op, the initial value of `x` also becomes the final value, and thus the initializer is not dead. More generally, conditional statements in Self, unlike languages with built-in conditional statements, cannot be recognized from the syntax alone[†]. To safely declare `x`'s initializer dead we must:

- determine a conservative approximation to all possible kinds of arguments that `five:` may be invoked on, and
- for each kind of argument prove that it either does not implement `ifTrue:False:` or implements it in a way that guarantees that at least one of the (block) arguments will be evaluated before `ifTrue:False:` returns.[‡]

[†] We use a conditional statement to illustrate the issues. However, other expressions that involve blocks, including loops, cause similar problems.

Accordingly, detection of dead initializers involves a non-local analysis. Worse yet, the analysis needs types—types that we want to compute *after* determining liveness of initializers. This circularity has shown up before, and so have two general solution techniques:

- *Iteration.* We can infer types assuming that no initializers are dead, then compute liveness information based on the inferred types, then use the liveness information to infer more precise types, then compute more precise liveness information, etc.
- *Integration.* We can integrate the two analyses that needs each other's results. To ensure monotonicity, initializers are assumed dead until proven otherwise. As the type inferencer builds the constraint graph and encounters more and more variable-reads, initializers may change monotonically from dead to live. When an initializer changes status, the associated variable's type may grow.

In the Self type inferencer, we reject the complexities of a non-local analysis to detect dead initializers. Instead, we conservatively assume that any block containing an assignment may not be executed, and any block containing a read may be executed. Under these assumptions, dead initializers can be recognized locally, although less precisely. For example, `x`'s initializer in `five:` cannot be deemed dead. Even so, the simple detection of dead initializers pays off sufficiently to be worthwhile for typical programs (see Table 19 on page 96).

Extrapolating the effect of dead initializer elimination from Self to another language may be difficult:

- *Lesser effect.* Dead initializers may occur with unusual frequency in Self because the language design encourages the programmer to create them. Initializers in Self methods are static: they are evaluated once when the method is created, rather than every time the method is invoked. The early and single evaluation makes initializers such as `|x <- set copy|` less useful (`x` will contain the same set every time the method is invoked). Instead, programmers write `|x|` and follow up with an assignment `x: set copy` in the beginning of the method, thereby creating a dead initializer. In languages with different initializer semantics, many dead initializers found in Self methods may not exist, reducing the importance of dead initializer elimination.
- *Greater effect.* The other language may have basic control structures built in. Thus, the local detection of dead initializers can be more effective, since common constructs such as conditional statements do not render it ineffective.

4.5.3 Static single assignment form

Static single assignment (SSA) transformation is a semantics-preserving program transformation that makes code more suitable for analysis [37, 78]. A program in SSA form assigns to each variable in one static location. The transformation introduces new variables and redirects accesses to the new variables. Figure 46 shows a simple Self method and the result of SSA transforming it. We will not attempt a complete SSA transformation on Self programs, but instead SSA transform methods individually and with respect to local variables only.

How does SSA transformation improve precision of type inference? The untransformed method in Figure 46 contains three expressions that read the variable `x` and three assignments to `x` (counting the initializer as an assignment). The type inferred for all the reads is $\{\overline{\text{nil}}, \overline{\text{smallInt}}, \overline{\text{float}}\}$, since at different points during the execution of the method, the variable `x` may contain `nil`, a smallInt, or a float. In this case, eliminating dead initializers does not improve the type of the reads, since `x` is read before the first assignment (at best, we could have hoped for the elimination of the `nil` initializer). In the SSA-transformed method, there are instead three different variables, $\mathtt{x}_0$, $\mathtt{x}_1$ and $\mathtt{x}_2$ (each assignment "introduces" a new `x` variable), and the three expressions that used to read `x` now read $\mathtt{x}_0$, $\mathtt{x}_1$, and $\mathtt{x}_2$. Consider $\mathtt{x}_1$ for a moment. Initially, it contains `nil`, but later a smallInt is assigned to it by $\mathtt{x}_1$`: 4`. The type of the variable $\mathtt{x}_1$ is therefore $\{\overline{\text{nil}}, \overline{\text{smallInt}}\}$. Moreover, $\mathtt{x}_1$ has a dead initializer, so after elimination of the initializer, the inferred type improves further to $\{\overline{\text{smallInt}}\}$. Thus, while the inferencer was unable to infer a better type than $\{\overline{\text{nil}}, \overline{\text{smallInt}}, \overline{\text{float}}\}$ for the second read of `x` in `straight`, it could infer the precise type $\{\overline{\text{smallInt}}\}$ for the corresponding read of $\mathtt{x}_1$ in `straight_ssa`. Table 20 shows how the types of all the variables improve in the SSA-transformed version of

‡ When the argument does not understand `ifTrue:False:`, we must additionally see if it understands `messageNotUnderstood` and verify that this method either aborts the computation or evaluates one of the blocks.

```
straight = (
  |x<-nil|
  x printLine.
  x: 4.
  x printLine.
  x: 5.5.
  x printLine.
  self.
).
```

SSA transformation $\longrightarrow$

```
straight_ssa = (
  |$x_0$<-nil. $x_1$. $x_2$|
  $x_0$ printLine.
  $x_1$: 4.
  $x_1$ printLine.
  $x_2$: 5.5.
  $x_2$ printLine.
  self.
).
```

Figure 46. A method before and after applying the SSA transformation

Both `straight` and `straight_ssa` have the same effect but the SSA version is easier to analyze precisely.

`straight`. In the following, we assume that the SSA transformation is always followed by dead initializer elimination to fully realize the benefits.

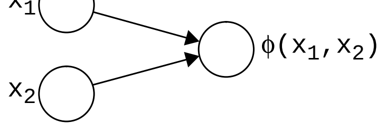
	Before SSA transformation (types in `straight`)		**After SSA transformation** (types in `straight_ssa`)	
Variable /access	**Before dead initializer elimination**	**After dead initializer elimination**	**Before dead initializer elimination**	**After dead initializer elimination**
x_0	{nil, smallInt, float}	{nil, smallInt, float}	{$\overline{\text{nil}}$}	{$\overline{\text{nil}}$}
x_1	{nil, smallInt, float}	{nil, smallInt, float}	{$\overline{\text{nil}}$, $\overline{\text{smallInt}}$}	{$\overline{\text{smallInt}}$}
x_2	{nil, smallInt, float}	{nil, smallInt, float}	{$\overline{\text{nil}}$, $\overline{\text{float}}$}	{$\overline{\text{float}}$}

Table 20. Types before and after SSA transformation of the `straight` method

Consider again the `five:` method and assume for now that the `ifTrue:False:` send implements a conditional statement. The SSA transformation introduces separate variables, x_1 and x_2, for each of the assignments of `x` in the two blocks. This creates a problem immediately after the conditional statement: is the current value of `x` found in x_1 or x_2? The problem is solved by inserting a ϕ *node*, $\phi(x_1,x_2)$, immediately after the conditional statement. The ϕ node performs a selection: depending on the branch along which control-flow enters the ϕ node, it evaluates to x_1 or x_2. With the use of a ϕ node, the SSA form of `five:` is:

```
five_ssa: arg = (
  |$x_0$ <- 'five'. $x_1$. $x_2$. $x_3$|
  arg ifTrue: [$x_1$: 5] False: [$x_2$: 5.0].
  $x_3$: $\phi$($x_1$,$x_2$).
  $x_3$.
).
```

In general, ϕ nodes must be inserted when control-flow merge points are reached by multiple definitions of a variable. At run-time ϕ nodes *select*; at type-inference-time they *merge*. The constraint generated for a ϕ node reflects the conservative assumption that control may have entered it through either branch:



The `five_ssa:` example demonstrates how ϕ nodes handle the merge of control flow for conditional statements. It is simple to generalize to other control structures, including loops. However, the problem remains that conditional statements (and other control structures) cannot be recognized syntactically in Self. Indeed, `five_ssa:` does not preserve the semantics of `five:` if the argument can be an object that implements `ifTrue:False:` as a no-op (in that case `five:` returns `'five'`, which `five_ssa:` can never return). The fault is not in the SSA transformation rules, but that we applied the SSA transformation rule for conditional statements to something that may not be a conditional statement. The problem can be overcome in two ways:

- *SSA-1*. We can perform a SSA-like transformation under sufficiently conservative assumptions about control flow. The transformation will be less powerful than if stronger assumptions had been made. For example, the conservatism means that "flow-around" cannot be ruled out in the `ifTrue:False:` send in `five:`, thus a three-way ϕ node must be used: x_3: $\phi(x_0, x_1, x_2)$.

- *SSA-2*. We can (attempt to) prove control-flow properties about sends before SSA transforming to permit stronger transformations. For example, after proving that the `ifTrue:False:` send in `five:` always evaluates one of the blocks, the stronger two-way ϕ node is valid. If the proof attempt fails, SSA-1 is a safe backup. Unfortunately, proving control-flow properties is difficult *before* type inference, but we could *iterate:* infer types, do SSA transformation, infer better types, do better SSA transformation, infer even better types, etc.[†]

Both SSA-1 and SSA-2 have disadvantages. SSA-1 is too pessimistic in the common case when `ifTrue:False:` implements conditional statements. SSA-2 incurs the high cost of a full type inference to do an SSA transformation. Interestingly, there is a third *integration*-like alternative:

- *SSA-3.* The type inferencer can perform an SSA transformation under the *optimistic* assumption that any send of `ifTrue:False:` implements a conditional statement (and use similar optimistic assumptions about other sends involving blocks). Upon completing inference, it examines the flow graph to determine whether the assumptions are valid (this examination is a "reaching definitions" problem). Whenever the inferencer finds an invalid assumption, it adds extra constraints to repair the problem (note: adding extra constraints preserves monotonicity). For instance, if it finds that the `ifTrue:False:` send in `five_ssa:` does not implement a conditional statement, it adds the constraint: type(x_0) $\subseteq$ type(x_3) to account for the fact that control may flow around both of the blocks in `five_ssa:`. When extra constraints are added, Step 3 of type inference (see Section 3.2.1) must be resumed to perform propagation and establish further constraints. When Step 3 finishes, the type inferencer performs another validation round, and so on. Termination and correctness follow from monotonicity.

SSA-3 covers a range of possibilities. The initial SSA-like transformation can be more or less optimistic, leaving less or more repair-work to the second phase. However, as the knob is turned towards more optimism and consequently more repairs in the second phase, the system becomes more and more like the one we describe in Section 4.5.4.

For Self, we have implemented SSA-1 with a few additional improvements, including:

- Placement of SSA-introduced variables inside blocks when possible, rather than at the same level as the original variable to maximize the effectiveness of dead initializer elimination.

- Placement of ϕ nodes inside blocks rather than outside, so that if the thread fails before reaching the ϕ node in some block method templates, it has no effect after the block.

- Suppression of ϕ nodes at exits of blocks with non-local returns.

The data in Table 19 on page 96 show that SSA-1 yields only a marginal improvement over the simpler dead initializer elimination. The conservatism of SSA-1 may be partly to blame. Another possible explanation may be that Self methods typically are very short. Thus, while even short methods benefit from the elimination of dead initializers, flow analyzing local variable accesses beyond the effect of the initializer gains little additional precision.

[†] While perhaps rare in practice, code sequences that defy iteration do exist. For example, even if a send implements a conditional statement, the *asymptotic precision* of the iteration of type inference and SSA transformation may be insufficient to prove this. The problem is that until we have proven that the send implements a conditional statement, we cannot SSA transform it as such.

In independent work, other researchers have also explored the use of SSA transformations to improve type inference precision. Plevyak and Chien translate Concurrent Aggregates programs into an intermediate form before analyzing them [94]. The translation includes an SSA transformation and a related SSU (static single use) transformation, which targets dependencies between multiple reads of a variable. Kumar, Agrawal, and Iyer also suggest combining SSA transformation and type inference [72] in the context of the BOBL language (see [88]). However, these papers do not discuss how the lack of control-flow information weakens the SSA transformation, possibly because both Concurrent Aggregates and BOBL, unlike Self, have common control structures built in.

4.5.4 Reaching definitions

The conservative assumptions necessitated by the lack of control-flow information limit the effectiveness of SSA transformation on Self methods. To gain further precision, the conservatism must be avoided or reduced. Instead of attempting to reason about control flow based on the mere syntax of the program, the reasoning should be based on the control-flow graph that the type inferencer builds. This way, more precise estimates of the flow of control from assignments to variable-reads is available to sharpen precision of type inference. A systematic way to exploit control flow information during type inference is:

- Compute reaching definitions (defined below).
- Define types for variable-reads based on reaching definitions.

A *definition* of a variable is either an initializer or an assignment of it. For example, in the `five:` method (shown again in Figure 44) there are three definitions of `x`: `x<-'five'`, `x: 5`, and `x: 5.0`. For a variable-read, the set of *reaching definitions* are those that may determine the value of the variable at that program point[†]. Thus, the set of definitions reaching a given program point *P* summarizes relevant properties of the control flow leading up to *P*. Exact computation of reaching definitions is impossible, but algorithms exist for computing conservative approximations.

Section 4.5.4.1 explains how to employ reaching definitions during type inference. Subsequently, Section 4.5.4.2 considers how to compute reaching definitions, with an emphasis on issues specific to the type inference context. Finally, Section 4.5.4.3 assesses the effectiveness of using reaching definitions.

Reaching definitions have not been incorporated in the Self type inferencer, so implementing and empirically evaluating the ideas outlined below is future work.

4.5.4.1 Using reaching definitions

The previous three approaches—the flow-insensitive analysis, the elimination of dead initializers, and the SSA transformation—all have in common that the type of a variable-read is determined from the accessed variable's type. In turn, the variable's type is determined as the union of the types of *all* definitions of the variable. Dead initializer elimination discounts initializers when they provably have no effect; SSA transformation introduces more variables so that each one has fewer definitions, and hopefully a smaller type.

Assuming that reaching definitions are available, more precise types can be inferred for variable-reads by parting with the principle that the accessed variable's type determines the type of the variable-read. Instead, the type of a variable-read can be based on the set of definitions reaching it:

- The type of a variable-read is defined as the union of the types of the definitions (of the relevant variable) reaching the read.

The type of the accessed variable no longer plays a role. For completeness, the type inferencer still infers a type for each variable, determined, as before, by all the definitions of the variable.

When the set of definitions reaching a given variable-read is a true subset of all definitions of the accessed variable, determining the type of the variable-read from the reaching definitions may yield a more precise type than without the

[†] Variable-reads are often referred to as *uses* and reaching definitions as *def–use chains*.

use of reaching definitions. As long as the computed set of reaching definitions conservatively approximates the exact set of definitions, the inferred type is sound. Figure 47 illustrates the use of reaching definitions for `five:` under the assumption that the set of definitions reaching the last expression `x` is {`x: 5`, `x: 5.0`}. Of course, the question remains as to how one would go about establishing that a definition such as the initializer `x<-'string'` does not reach a variable-read.

```
five: arg = (
  |x<-'five'|
  arg ifTrue: [x: 5] False: [x: 5.0].
  x.
).
```

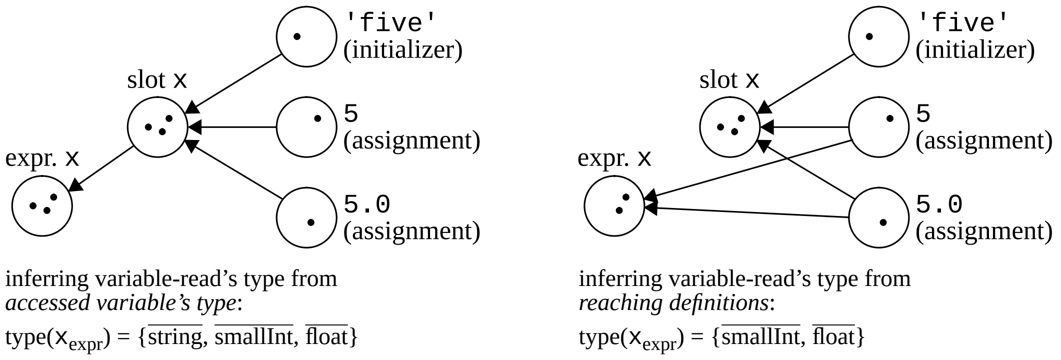

'five'
(initializer)
slot x
5
(assignment)
expr. x
5.0
(assignment)
inferring variable-read's type from
accessed variable's type:
type(x_{expr}) = {$\overline{\text{string}}$, $\overline{\text{smallInt}}$, $\overline{\text{float}}$}
'five'
(initializer)
slot x
5
(assignment)
expr. x
5.0
(assignment)
inferring variable-read's type from
reaching definitions:
type(x_{expr}) = {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}

Figure 47. Improving type inference with reaching definitions

When reaching definitions are used to infer the types of variable-reads, only a subset of all definitions of the variable influence the type of the read, thus yielding smaller and thereby more precise types.

4.5.4.2 Computing reaching definitions

Type inference computes a global control-flow graph in which the templates represent method invocations. Once a control-flow graph is available, existing algorithms for interprocedural computation of reaching definitions can be applied [52]. In general, these algorithms propagate definitions forward through the flow graph:

- When a definition is propagated into a program point that assigns to the same variable, the second definition *kills* the former and is the only one propagated out of the program point (assuming the second definition is a strong update [29]; if it is a weak update, both definitions survive the encounter).

- At points where control flow merges, the union of the incoming definitions is computed and propagated out of the merge point.

To ensure precision, the propagation should be limited to *valid paths* [89, 99, 105]: the subset of all paths in the control-flow graph that respect the restriction that call/return edges pair up correctly. We refer the reader to the literature for general aspects of these algorithms. There are, however, specific issues that must be considered when adapting an algorithm to constraint-based analysis for object-oriented programs:

- *Polyvariance*. Precise type inference requires a polyvariant analysis, i.e., an analysis that may create more than one template for each method. We expect the same to be true for reaching definitions. Fortunately, the templates created by a polyvariant type inferencer readily supply the skeleton for a polyvariant computation of reaching definitions: instead of propagating definitions through a structure derived from the program's syntax, they should be propagated through a structure derived from the collection of templates.

- *Iteration or integration.* Reaching definitions and type inference are mutually dependent problems: reaching definitions can improve type inference, but cannot be computed without the control-flow graph built by the type inferencer. The two standard solutions seem applicable.

  We can *iterate*: infer types flow-insensitively, compute reaching definitions, infer more precise types using the reaching definitions, compute better reaching definitions, infer even more precise types, etc. The initial hurdle seems passable, but it is easy to find code examples with inferior asymptotic precision (e.g., a variable access where the assumption that a certain definition is reaching creates a flow path that makes it reachable).

  We can *integrate* type inference and computation of reaching definitions. Correctness requires that types of variable reads grow monotonically during the combined analysis. Since type inference (inductively) guarantees that the type of each definition grows monotonically, overall monotonicity is ensured if the set of definitions reaching each read grows monotonically during analysis. Initially, no definitions reach any use since the initial control-flow graph has no edges. As edges are added during inference, definitions can flow on more valid control flow paths. In order to establish that the set of reaching definitions grows monotonically for each read, we must argue, for the specific reaching definitions algorithm, that the addition of an edge to the control-flow graph cannot prevent a definition from reaching a read that it could reach before the edge was added. Finally, to ensure efficiency, an incremental reaching definitions algorithm is preferable to avoid having to recompute the reaching definitions from scratch each time the type inferencer adds an edge to the control-flow graph.

Whether the iterative or integrated approach for computing reaching definitions is preferable in a given situation depends on several factors, including: does the type inferencer already iterate for some other reason? How efficient is the incremental reaching definitions algorithms used in the integrated approach compared with the non-incremental algorithm that suffices for the iterative approach? For the Self type inferencer, we are inclined to favor the integrated approach. However, we have no empirical data yet to substantiate this expectation.

4.5.4.3 Assessing reaching definitions

We have not added computation of reaching definitions to the Self type inferencer, and thus cannot give empirical data on its effectiveness. Instead, we give three concrete examples where reaching definitions can improve precision over the SSA-1 transformation. All three examples concern the type inferred for the read of `x`, in the `five:` method. Recall that the SSA-1 based inferencer infers the type {$\overline{\text{string}}$, $\overline{\text{smallInt}}$, $\overline{\text{float}}$} because it cannot rule out that flow may bypass both branches of the "conditional." Now consider the use of reaching definitions:

- If the target program invokes `five:` with `true` or `false`, all paths from the initializer `x<-'five'` to the read of `x` pass through either `x: 5` or `x: 5.0`. Consequently, the initializer is always killed, the set of definitions reaching the read is {`x: 5`, `x: 5.0`}, and the type is {$\overline{\text{smallInt}}$, $\overline{\text{float}}$}.

- If the target program invokes `five:` only with `true`, all paths in the control-flow graph from `x<-'five'` to the read pass through the definition `x: 5`. Hence, the only definition reaching the read is `x: 5`, and the type is {$\overline{\text{smallInt}}$}.

- If the target program invokes `five:` only with the object that implements `ifTrue:False:` as a no-op, the control-flow graph has a path from the initializer `x<-'five'` to the read, but no path through the two definitions in the blocks, since the blocks are never invoked. Thus, the type is {$\overline{\text{string}}$}.

In conclusion, reaching definitions have the potential to improve the precision of types inferred for variable-reads more than the other flow-sensitive techniques. We have suggested two approaches to combining reaching definitions with constraint-based analysis. The iterative approach is simpler to implement; the integrated approach may be more efficient and precise.

4.6 Grouping—how to analyze a heap of objects

Recall from Section 2.2.2 that the Self 4.0 programming environment operates with objects as the primary representation of programs, and emphasizes direct interaction with objects. To support this model of programming, the type inferencer also operates directly on objects. It analyzes the objects themselves, not the code that created them. Consequently, there is a need to handle large numbers of objects: unlike programmer-written source code which tends to be compact and non-redundant[†], the Self image can be more repetitive and redundant. For example, if an application needs fast access to all prime numbers less than 100,000, the programmer has two options:

- Hardwire the prime numbers into the program:

  ```
  primes = 100000 asVector filterBy: [|:i| i isPrime] Into: set copy.
  ```

  The above line creates a set with all prime numbers less than 100,000 and stores it in the slot `primes`. The prime numbers are computed *before* program execution (during the parsing phase) and are used in all executions of the program.

- Use an initialization method that computes the prime numbers during startup of the program:

  ```
  primes. "Set of prime number will be stored here."
  ...
  initialize = (
    primes: 100000 asVector filterBy: [|:i| i isPrime] Into: set copy.
  ).
  ```

While the two alternatives may look similar in print, they differ significantly for type inference. In the former case, the type inferencer must analyze a set with 9592 different smallInts (the expression that created the set may no longer be around); in the latter case, it must analyze a method that creates this set.

We have investigated *grouping* as a way to maintain high efficiency of type inference when large numbers of objects must be analyzed. Section 4.6.1 introduces grouping and an important trade-off between precision and efficiency it must make. Section 4.6.2 presents the specific way the Self type inferencer groups objects. Finally, Section 4.6.3 discusses shortcomings, alternatives, and possible improvements.

4.6.1 Trading analysis time against precision

To ensure the efficiency of type inference, redundant work must be avoided. We have designed and implemented *grouping* of initial objects as a way to avoid redundancies arising from similar objects. Grouping can be viewed as a front-end to the type inferencer. Instead of allowing the type inferencer direct access to individual objects, the front-end partitions objects into groups according to how “similar” the objects are and presents the type inferencer with a grouped view of the image. For example, all smallInt objects form a single group. Other groups are bigInts, points, and true (this group has only one member, the object `true`). In our implementation, groups are built on demand to avoid spending resources grouping objects to which the type inferencer never needs access.

Grouping gains efficiency because it exempts the type inferencer from analyzing many similar objects individually. Instead, the inferencer can analyze the whole group at once. Grouping sacrifices precision since the type inferencer cannot distinguish individual members of the same group. Thus, to ensure a reasonable balance between efficiency and precision, the granularity of the groups must be chosen with some care. To illustrate, consider these two extremes:

- *Minimal resolution.* If there is only one group, OBJECT, that contains all initial objects, type inference is very fast since the only type is $\{\overline{\text{OBJECT}}\}$. However, type inference then computes only trivial information.

- *Maximal resolution.* If every initial object is in its own group, type inference will be slow since there are many initial objects. However, type inference will compute very precise information.

Although each extreme is undesirable, reasonable compromises exist. The next section outlines a particular grouping strategy that has been successfully applied to Self.

[†] Machine generated programs need not share this property.

4.6.2 Grouping rules

We have informally characterized groups as sets of sufficiently similar objects. It is now time to be more precise about what “similar” means. First, consider the rules that define grouping for “primitive” objects:

- All smallInts form a group.
- All floats form a group.
- All canonical strings[†] form a group.

The above rules can actually be derived from the set of more general rules described below. However, we have pulled them out because they give a good intuition about what grouping does and the trade-offs involved. Consider the first rule. It captures a *decision* not to distinguish between different smallInts. This decision is not the only one which would work. For example, if a particular use of type information could benefit from more precise information about 0 and ±1, we could modify the grouping of smallInts to give 0 and ±1 their own groups. With this finer grouping in place, the specification of the various smallInt primitives could be refined to extract and preserve as much knowledge as possible. For example, the integer multiplication primitive returns the 0-group if and only if one of its arguments is the 0-group.

Before describing the rules for grouping non-primitive objects, we need to explain that there are several *categories* of objects in the Self system. In addition to “regular” objects, there are proxies, function proxies, vectors, byte vectors, mirrors, smallInts, floats, canonical strings, profiler-objects, methods, blocks, block methods, processes, and stack frames (activation records). Since the different categories of objects are often treated differently by the virtual machine, we also want the type inferencer to distinguish between them. With this explanation in place, we put two objects in the same group, unless doing so would violate one or more of the following rules:

- Rule 1 (structure). Two objects can be in the same group only if they are of the same category.
- Rule 2 (structure). Two objects can be in the same group only if they have the same set of slot names and pair-wise the slots are both parents or both non-parents.
- Rule 3 (slot contents). Two objects can be in the same group only if the contents of their constant slots pair-wise belong to the same group[‡].
- Rule 4 (mirrors). Two mirrors can be in the same group only if their reflectees are in the same group[††].
- Rule 5 (vectors). Two vectors can be in the same group only if the *sets* of groups of their elements are equal. For example, two vectors of strings belong to the same group, but a heterogeneous vector of integers and strings is in a different group.
- Rule 6 (methods). Each method is always in a group of its own (consequently, by Rule 3 each block is always in a group of its own, since its block method resides in a constant slot).

[†] The Self system defines several variants of character strings. Canonical strings result from evaluating literal string constants such as `'guitar string'`, but can also be created by converting (canonicalizing) other kinds of strings. Canonical strings are immutable and guaranteed unique: no two of them can have the same characters unless they are the same object (like symbols in Smalltalk).

[‡] Self slots may be either constant or assignable. The contents of constant slots cannot be changed by assignment, although they can be changed by reflective operations. The distinction between constant and assignable slots is orthogonal to the distinction between parent and non-parent slots.

[††] Mirrors implement structural reflection in the Self system. Any object can be “reflected” to obtain a mirror object. The mirror defines a protocol that allows arbitrary modifications of the reflectee, including adding and removing slots, changing slots from being parents to non-parents and vice versa, and changing the contents of constant slots [4].

The group of an object relies recursively on the groups of the objects in its constant slots (Rule 3). The possibility of cyclic object structures makes it complicated and costly to implement of this rule. To avoid these complexities, the implemented system uses a simplified version of the third rule:

- Rule 3′ (slot contents). Two objects can be in the same group only if the contents of their constant slots are pair-wise identical or contain the same kind of primitive objects (integers, floats, and canonical strings).

Since the modified rule is no longer recursive, the test is easy to implement and executes quickly. For most programs, Rule 3 and Rule 3′ differ only marginally: the latter may result in slightly more groups being created, thus slowing down type inference a bit.

We have tried to keep the grouping rules simple, sometimes at the cost of information loss. Possible improvements to the rules and the grouping issue as a whole are discussed in the next section. To illustrate the rules, here are some examples of objects that are in the same group:

```
(| parent* = traits oddball. x <- 8.   |)
(| parent* = traits oddball. x <- 9.   |)
(| parent* = traits oddball. x <- nil. |)
```

The last object is in the same group as the first two because the grouping rules ignore the contents of assignable slots, except for vector elements. This is one source of precision loss; see Section 4.6.3.2. For another example, these objects are in different groups:

```
(| parent* = traits clonable. doThat = ( 4 do: [that]. ). |)
(| parent* = traits clonable. doThat = ( 9 do: [that]. ). |)
```

The reason is that they contain different methods (which are in different groups) in the constant slot `doThat`. Finally, of the four vectors resulting from these expressions, the first two belong to the group of "vectors of smallInts" and the last two belong to the group of "vectors of smallInts or floats":

```
vector copySize: 9 FillingWith: 47.
vector copySize: 8 FillingWith: 14.
(vector copySize: 7 FillingWith: 8  ) at: 3 Put: 4.5.
(vector copySize: 6 FillingWith: 3.5) at: 2 Put: 6.
```

4.6.3 Discussion

The rules given above produce a reasonable trade-off between cost and precision of type inference for many programs, including the benchmarks described in Section 3.4. There are, however, several ways in which grouping can be improved to provide better precision and/or efficiency of type inference. In this section, we discuss some of the more interesting options. We also consider how grouping fits into a programming environment where objects—potentially, at least—change all the time. Finally, we relate groups to classes.

4.6.3.1 Regrouping

When an object is first encountered by the inferencer, it is put into a group. To decide which group an object belongs to, the contents of its slots are inspected (other factors that may be involved are discussed below). Since programming may change objects arbitrarily, it may also invalidate grouping. For example, these objects

```
(| parent* = traits oddball. x <- 8. |)
(| parent* = traits oddball. x <- 9. |)
```

are in the same group, but if a programming action changes the second object to

```
(| parent* = traits oddball. x <- 9. y <- 1. |)
```

the two objects no longer belong to the same group. Thus, to maintain a valid grouping, it is necessary to regroup selected objects after certain changes. While we have not implemented a system for selective regrouping, it should be possible to do so by inserting hooks into the (reflective) programming methods so that programming actions automatically trigger regrouping of the affected objects.

4.6.3.2 Assignable slots

The grouping rules given above ignore the contents of assignable slots. The result may be loss of precision because objects with different contents in assignable slots may be merged. For example, the “link” objects from a list of integers may be grouped with the link objects from a list of floats, making both lists appear as heterogeneous lists of integers and floats. To prevent this inaccuracy, Rule 3 can be modified to apply to all slots:

- Rule 3″ (slot contents). Two objects can be in the same group only if the contents of their ~~constant~~ slots pair-wise belong to the same group.

The modification improves precision, but has its own problems:

- A simple assignment may now make regrouping necessary. In programming environments with less emphasis on direct manipulation of objects, or if type inference supports compilation or extraction of final applications (i.e., applications that are no longer changing), this vulnerability of grouping may not be a problem.
- The trade-off between cost and precision may be pushed too far in the direction of precision. Consider the linked list of integers in Figure 48. Rule 3″ forces each link object into its own group. The links containing `4` and `5` are in different groups because the former link’s `next` slot contains a link object whereas the latter link’s contains `nil`. Backtracking one step, since the 4-link and the 5-link are in different groups, so are the 3-link and the 4-link. Repeating this argument, it can be seen that all the links in the list are in different groups. Consequently there are many groups and analysis becomes costly.

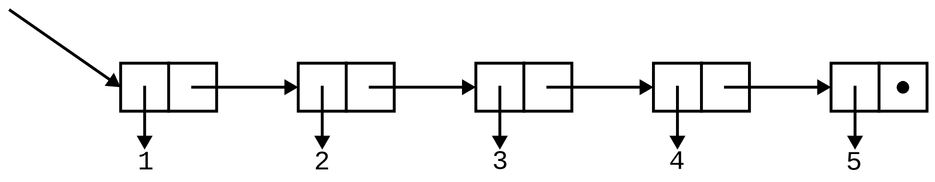


Figure 48. A linked list can be a challenge for grouping

To overcome these problems, it may be necessary to base grouping on more than the state of objects. This issue is discussed in general in the next section. Here we conclude with an example specifically related to assignable slots. In statically-typed languages, it is possible to compare the static type declarations of assignable slots when grouping objects. For example, consider these two Beta lists†:

```
List(# element :< Figure #)        (* List of figures: circles, squares, rectangles, ... *)
List(# element :< Square #)        (* List of squares. *)
```

The declared type of the elements in the second list is more specific than in the first list, suggesting that the two lists should be assigned to distinct groups. We can phrase this idea as follows:

- Rule D (declarations). Two objects can be in the same group only if the statically declared types of their assignable slots are pair-wise the same.

This rule will force a list of figures into a different group than a more specific list of squares, thus improving precision of type inference in the parts of the program that deal with the more specific list.

† The syntax “`:<`” is not a special-purpose construct for defining parameterized collection classes; it denotes the further-binding of a virtual pattern; see [75].

4.6.3.3 Feedback from type inference to grouping

To improve grouping, it seems necessary to go beyond a scheme based exclusively on the state of the object. Fortunately, at the point when the type inferencer needs the group of an object, more than the state of the object is known:

- The type inferencer has information about how the application accesses the object, e.g., through which slot the object may be fetched.
- The type inferencer may have information about what will happen to the object in the near "future" in terms of the flow paths that the inferencer is tracing.
- The type inferencer may be able to determine that the application does not access a specific slot of an object. Hence, this slot need not be considered when grouping the object. Along the same lines, a coarse grouping can be used initially and gradually refined as the need is seen. The initial grouping can be based on the assumption that *no* slot will be accessed in the objects. Whenever the type inferencer analyzes an expression that accesses a slot, the group can be split into a number of smaller groups, each of which contains the objects from the original group that have similar contents of the accessed slot.

Tapping into these additional sources of information may improve grouping and therefore increase precision and efficiency of type inference. It may also make the grouping more vulnerable and necessitate more frequent regrouping of objects. Overall, the interface between the type inferencer and the objects in the image holds promise for future work.

4.6.3.4 Classes versus groups

Grouping plays an important role in the Self type inferencer, but what if the goal is to analyze a different, class-based language? In a class-based language, an obvious approach is to base grouping on classes:

- Rule C (classes). Two objects can be in the same group only if they are direct instances of the same class.

This rule can replace the structural comparison rules given in Section 4.6.2 (Rule 1 and Rule 2). However, Rule C alone cannot replace all six rules of Section 4.6.2. Consider Smalltalk for a moment. All Smalltalk vectors are instances of the same class. Thus, Rule 5 (the vector rule) is still needed unless we want all vectors in the image to be in the same group. Hence, even though classes may simplify grouping, in general they cannot fully replace grouping.

4.7 Incremental type inference

Fast type inference, while always desirable, is *essential* for some applications. In this section, we propose a simple incremental analysis technique that can speed up type inference when analyzing a sequence of programs that share code. For example, a type-inference-based browser can exploit this notion of incrementality. In a typical program development situation, the programmer repeatedly subjects different versions of the same program to type inference, because the type inference step is inserted in the edit-compile-run loop (or the edit-continue loop of the Self system).

The incremental type inferencer exploits the similarity between successive target programs to speed up type inference. It gains speed by reusing templates from one analysis to the next. Continuing with the browsing and editing example, when the programmer modifies a method in the program, the incremental type inferencer discards the method's templates, but retains all other templates. Now, the next program version can be analyzed faster because many of the required templates already exist. If the program is changed only slightly, few templates need to be flushed and the incremental analysis is very fast. As the changes become more extensive, the incremental analysis takes longer. At some point the changes are large enough that the complete analysis can be performed faster than the incremental analysis.

The effect on the inferred types of a local change in a method can vary significantly. Consider these three cases, which are extreme in the sense that they involve the smallInt addition method upon which many other methods depend:

- *Neutral.* If the programmer modifies the smallInt addition method to print out a warning message when an overflow occurs, the incremental analyzer can analyze the modified program very quickly: since the interface of the method remains unchanged, the incremental analyzer only needs to create templates for the new smallInt addition method.

- *Adding.* If the programmer modifies the smallInt addition method to return a new kind of object, say an "overflow" object, the incremental analyzer has more work to do: it must extend the previous analysis in all the places that use the result type of the smallInt addition method. This change, however, is straightforward, since the new result type is larger, i.e., preserves monotonicity of type inference.
- *Removing.* A modification that removes object types from the result type of a method (e.g., disabling bigInt arithmetic removes $\overline{\text{bigInt}}$ from the result type of the smallInt addition method) can be the hardest kind of change to handle for the incremental analyzer. Monotonicity restricts the type inferencer in this case. It cannot straightforwardly remove object types from the existing constraint graph. It can, however, flush and recreate the templates of the modified method and all templates that transitively depend on the result types of the modified method. A less time-consuming alternative is to leave the "too large" types in the network and accept that the available type information is less precise (it is still sound) until the next complete type inference.

Speed-up from reusing templates is not restricted to different versions of the same program. The same technique applies *across* programs, improving performance when code shared between the programs need only be analyzed once. Consider, for instance, bigInt arithmetic. The first program that uses arithmetic forces an analysis of the bigInt implementation. Subsequent programs can be analyzed faster because they can be connected to arithmetic templates that were created during the first analysis.

Code libraries are obvious targets for the incremental analysis technique. They can be shipped with a set of templates that partially describe typical uses of the code. To analyze a client program, the type inferencer can use the pre-built templates to the extent they match the client programs interactions with the code library. Self's bigInt implementation is an example of a code library where this idea has a high pay-off: shipping the bigInt implementation with pre-built templates for the arithmetic routines can speed up the analysis of virtually all Self programs.

4.7.1 Empirical results

We have implemented a restricted version of the incremental analysis idea. The implementation does not support selective flushing of templates, but does permit programs to be analyzed "in sequence" where each analysis benefits from the templates created during the analyses of the previous programs. Figure 49 shows the results of applying this limited incremental analyzer to Factorial and Richards. The figure should be read starting in the top left corner and following the arrows along the two possible paths to the bottom right corner. The top left corner represents the initial situation in which no program has been analyzed. The horizontal arrow out of this corner represents a full (non-incremental) analysis of Factorial, taking 6.8 CPU seconds and creating 6,208 templates. Subsequently, as denoted by the right-most vertical arrow, the incremental analyzer can analyze Richards in just 0.8 seconds by creating an additional 516 templates. Likewise, the vertical arrow out of the top left corner represents a full analysis of Richards, taking 7.2 seconds and creating 6,717 templates. Following the analysis of Richards, the bottom-most horizontal arrow shows that the incremental inferencer can analyze Factorial in 0.3 seconds by creating 7 additional templates. The much faster times for the incremental analyses, for Factorial 0.3 seconds versus 6.8 seconds, and for Richards 0.8 seconds versus 7.2 seconds, reflect the incremental analyzer's ability to take advantage of templates existing from previous analyses.

In the present example, whether the sequence of programs analyzed is Factorial-Richards or the reverse Richards-Factorial, the type inferencer creates the same number of templates:

$$6{,}208 + 516 = 6{,}724 = 6{,}717 + 7.$$

This property, however, does not hold true in general, since the incremental analyzer's inability to "undo" assignments to instance variables shared by two programs may perturb the numbers (usually only to a small degree).

To sum up, while the implemented incremental analyzer lacks important features such as selective invalidation of templates, it does support our claim that by reusing templates from analysis to analysis, type inference of a sequence of programs sharing code can be performed significantly faster than if each program is analyzed separately.

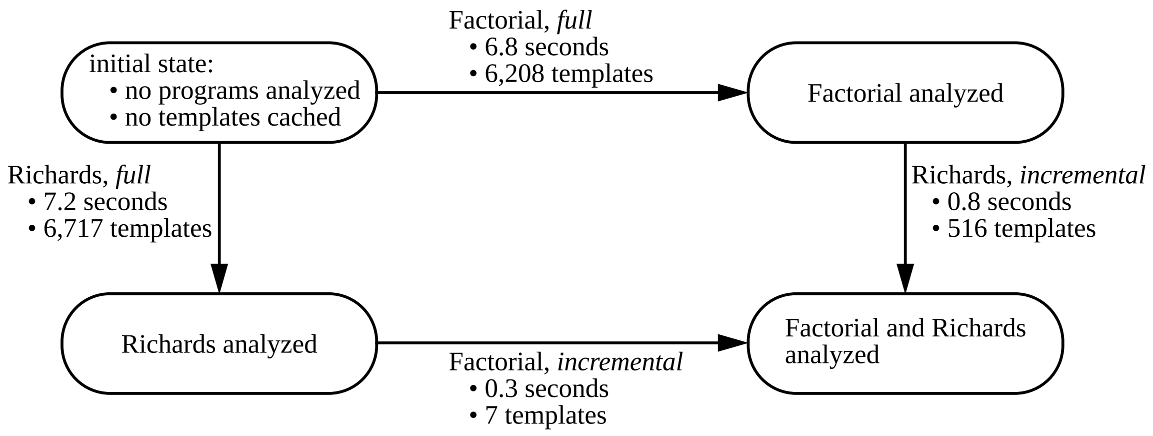


Figure 49. Demonstration of speed-up achievable by incremental type inference

The upper path from the top left corner to the bottom right corner shows measurements obtained on a full analysis of Factorial followed by an incremental analysis of Richards. Along the lower path, the two programs are analyzed in the reverse order, starting with a full analysis of Richards followed by an incremental analysis of Factorial.

4.8 Summary

Several important issues must be considered to get from an algorithm for analyzing code with parametric polymorphism to a complete type inference system:

- *Dynamic dispatch resolution.* Suzuki employed an iterative resolution algorithm. Palsberg and Schwartzbach suggested an integrated approach, based on encoding degenerate lookup rules as conditions over the inferred types. We generalized their approach to support multiple and dynamic inheritance in a natural way, and in the process factored the lookup rules of the analyzed language from the rest of type inference. Our analyzer employs a lookup algorithm, derived directly from the normal run-time lookup algorithm, during type inference.

- *Inheritance.* Previous systems expand away inheritance before type inference to boost precision. In contrast, the cartesian product algorithm can analyze unexpanded code with the same precision, making it a particularly attractive choice when analyzing languages with powerful inheritance schemes and/or multiple dispatch.

- *Data polymorphism.* Both parametric and data polymorphism impact precision of type inference. The principal contribution of this dissertation, the cartesian product algorithm, addresses only parametric polymorphism. We extended the framework for constraint-based type inference to accommodate the existing algorithms for analyzing code with data polymorphism. These algorithms improve precision by splitting object types, and can be combined with the cartesian product algorithm.

- *Blocks.* Blocks occur frequently in Self programs. Blocks may access variables in their lexical scopes and perform non-local returns. Both features require special attention during type inference to simultaneously secure precision and efficiency. To obtain precision when analyzing lexically-scoped variable accesses and non-local returns, we introduced closure object types and added lexical pointers to templates. Both of these ideas directly reflect execution-time concepts. To obtain efficiency, we proposed fusing similar closure object types.

- *Grouping.* To permit efficient type inference directly on objects rather than on a source representation of programs, the Self type inferencer employs grouping. Grouping trades precision against efficiency.

- *Flow-sensitive analysis of variable accesses.* A flow-sensitive type inference algorithm infers more precise types for variable-reads than a flow-insensitive algorithm by conservatively assessing the sequencing of assignments and reads of variables. We presented three increasingly powerful approaches for combining flow sensitivity with constraint-based type inference: dead initializer elimination, SSA transformation, and employment of reaching definitions.

- *Incrementality.* A complete reanalysis of the target program in response to a small change in it would waste effort. Instead, the existing type information should be repaired where necessary and left unchanged elsewhere. We proposed a simple technique to achieve this incrementality of type inference: reuse of templates from analysis to analysis. Our technique applies not only to versions of the same program, but also to different programs that share code. Its effectiveness was demonstrated experimentally.

Throughout, we have combined mutually dependent analyses, one of which is type inference (the other can be dynamic dispatch resolution, the sharing decision of adaptive type inference, splitting of clone families to resolve data polymorphism, or various forms of flow analysis of variables). We characterized two general approaches, *integration* and *iteration*, for combining analyses and explained how they apply in several specific cases. The approaches appear to have complementary strengths and weaknesses for many specific analyses.

We have discussed important issues that arose in our work on type inference for Self. Most of the issues are also relevant for other languages and programming environments, although no language would involve the exact same set of issues. By viewing the issues as a collection of case studies, we hope other researchers may find inspiration in them when tackling problems that are specific to their language or environment.

5 Recursive customization

Recursion, with deep roots in mathematics, is today a routine technique for concisely expressing solutions to many programming problems[†]. Consequently, we want to analyze recursive methods efficiently and precisely. However, recursion challenges adaptive type-inference algorithms, including the cartesian product algorithm, by potentially causing non-termination of analysis. Recursion may also affect optimizing compilation, as will be demonstrated in Section 5.1.2. Informally, non-termination happens when the creation of new templates results in the creation of new closure object types that in turn necessitate the creation of new templates, etc. A simple recursive method that invokes itself with a closure of its own demonstrates this problem:

```
rec: blk = ( self rec: [blk value] ).
```

Each `rec:` template the type inferencer creates contains a send (the recursive one) that invokes `rec:` with a new closure object type. This send therefore demands the creation of a new `rec:` template and so on. We refer to this situation as *recursive customization*. More complicated cases involve indirect recursion; we later present an example of 23-way indirect recursion.

To ensure termination, recursive customization must be broken by introducing a cycle in the constraint graph. The cycle connects a recursive send to a previously created template for the invoked method, making it possible to analyze the send without creating a new closure object type. The cycle should be introduced judiciously, else precision loss may result (just as the basic algorithm loses precision because it merges too many cases). For example, the pessimistic approach of treating every send as recursive falls short. Instead, we apply a conservative test to detect situations where recursive customization may occur, and only then direct the type inferencer to proceed with more caution, i.e., customize less to maintain termination guarantees. Hence, we consider two subproblems related to recursion:

- *Detection*. Detect cases where recursive customization may happen.
- *Avoidance*. Infer types for these cases in a way that ensures termination.

The remainder of this chapter is organized as follows. Section 5.1 studies the recursive customization problem in detail. Section 5.2 describes how we solved the avoidance subproblem; it is the simpler of the two, so we attack it first. In turn, Section 5.3 presents solutions to the detection subproblem. Finally, Section 5.4 summarizes the chapter. Throughout this chapter, we assume the cartesian product algorithm is used, but with some modifications, the results apply to adaptive type-inference algorithms in general.

5.1 Understanding recursive customization

To effectively counter the recursive customization problem, it must be understood in detail. The problem is not trivial, involving an interaction of closures, recursion, and customization. Indirect recursion may further complicate matters. This section aims to develop a better understanding of recursive customization. Section 5.1.1 focuses on the interaction of closures and recursion that leads to recursive customization. Section 5.1.2 documents that recursive customization goes beyond type inference: the Self compiler faces the same problem when compiling customized methods. Finally, Section 5.1.3 looks at indirect recursion.

5.1.1 The role of closures

Although type inference of recursive methods *may* diverge due to recursive customization, many recursive methods can be analyzed without encountering problems. Consider the recursive factorial function and a send invoking it:

[†] Recursion was not always routine. In his Turing Award Lecture [56], Hoare recollects inventing QUICKSORT in 1960, but only later realizing that it could be elegantly expressed recursively: "… It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded."

```
fact = (
  0 = self ifTrue: [^ 1].
  self * (self - 1) fact.
).
```

`5 fact printLine.` "Invoke the `fact` method on `5` and print the result."

When analyzing the send expression `5 fact`, the type inferencer creates a template for `fact` in which type(`self`) = {$\overline{\text{smallInt}}$}. In this template, the recursive send expression `(self - 1) fact` has receiver type {$\overline{\text{smallInt}}$}[†], so the inferencer connects it to the same template as it is found in; see Figure 50. Thus, type inference terminates after analyzing a single `fact` template — despite the recursion.

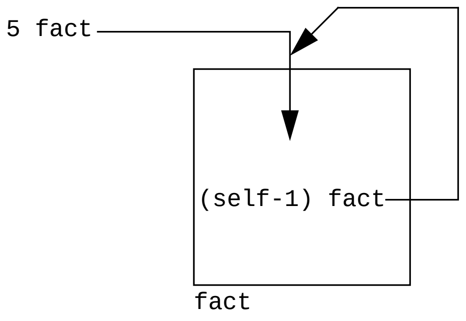


Figure 50. Recursion, as in the `fact` method, need not imply recursive customization

The recursive send `(self - 1) fact` has the same receiver type as the original send, `5 fact`. Hence, the cartesian product algorithm connects both sends to the same template, creating a cycle in the call graph. The cycle finitely represents an unbounded number of calls that may result from executing `fact`, and ensures the termination of type inference.

In contrast to `fact`, the following recursive continuation-passing-style (CPS) factorial method causes problems for the type inferencer (for the history of continuations, see [100]).

```
factCPS: resultBlk = (
  0 = self ifTrue: [^ resultBlk value: 1].
  self - 1 factCPS: [|:a| resultBlk value: a * self].
).

(6 factCPS: [|:a| a]) printLine.
```

Type inference of `6 factCPS: [|:a| a]` does not terminate. Figure 51 shows what goes wrong: the type inferencer connects each recursive call to a new template. In more detail, to analyze the first send of `factCPS:`, it creates a template in which the receiver's type is {$\overline{\text{smallInt}}$} and the argument's type is the "print result continuation (closure)," `[|:a| a]`. Processing this template, the type inferencer encounters the second call of `factCPS:`, the recursive call

```
factCPS: [|:a| ^ resultBlk value: a * self].
```

In this call, the receiver is again a smallInt, but the argument is a new closure. Thus, the inferencer connects the second call to a different (new) template than the first call. The new template contains a third call of `factCPS:`. Although the arguments of the second and third calls are both closures of

`[|:a| ^ resultBlk value: a * self]`,

[†] Strictly speaking, the receiver in the recursive call has type {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}, resulting in two outgoing edges from the recursive call. However, to simplify the examples, we ignore arithmetic overflows in this chapter.

they have different object types, since their respective environments bind `resultBlk` to closures with different object types. Consequently, the second and third calls must be connected to different `factCPS:` templates. This pattern repeats indefinitely and the analysis of `factCPS:` does not terminate.

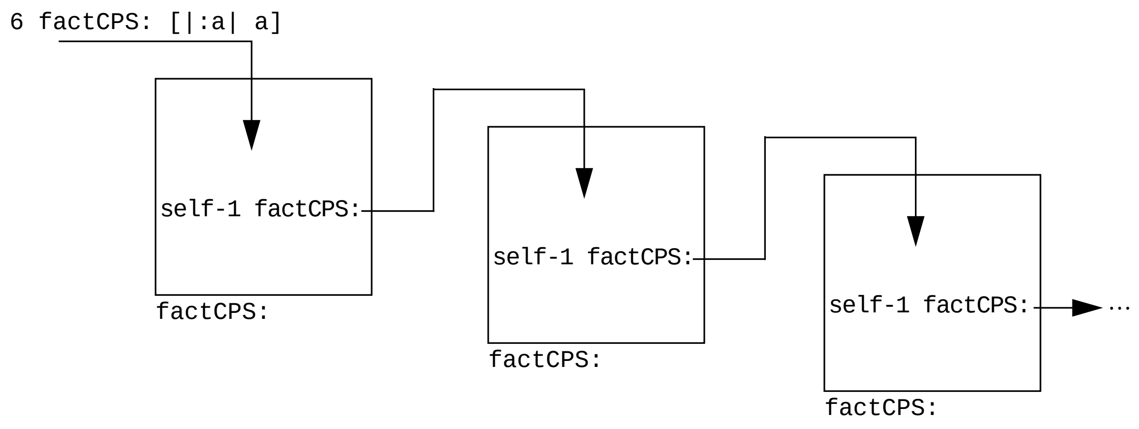


Figure 51. Recursive customization in `factCPS:` method

Each time a recursive send of `factCPS:` is analyzed, its argument is a new closure object type. The type inferencer therefore connects the call to a new template, creating an unbounded chain of `factCPS:` templates. Consequently, type inference does not terminate.

Having seen the first detailed example of recursive customization, we recapture the informal definition given in the introduction:

Definition. *Recursive customization* refers to the situation where a recursive method cannot be analyzed in finite time, because each time the type inferencer analyzes a template with a recursive call, it must create a new template that also contains a recursive call.

We have seen that `fact` can be analyzed straightforwardly whereas `factCPS:` cannot because it results in recursive customization. In general, a necessary condition for recursive customization is that the inferred type of an actual argument of the recursive call contains a closure object type. If none of the actual arguments have a closure object type in their type, only a finite number of object types remain as possible arguments. Consequently, only a finite number of templates are required for methods invoked by recursive sends, in turn ensuring the termination of type inference[†].

The necessary condition for non-termination of type inference can, of course, also be phrased as a sufficient condition for termination: “no recursive call passes a closure (according to the inferred types).” This observation can be used to argue that the analysis of `fact` terminates, since `fact` satisfies the condition that no recursive call passes a closure. However the condition is stronger than necessary: there are recursive methods whose recursive calls pass closures, yet can be analyzed in finite time. For instance, imagine adding an extra tracing argument to `fact` to debug it:

```
factTrace: traceBlk = (
  traceBlk value: self.
  self = 0 ifTrue: [^ 1].
  self * (self - 1 factTrace: traceBlk).
).

'9! = ' print.
(9 factTrace: [|:n| 'fact invoked on ' print. n printLine]) printLine.
```

[†] Although formal proofs could be derived, we feel that the added insight they provide in this case does not carry the weight of formalizing the context.

`factTrace`: passes a closure in every recursive call, but since the recursive call passes the exact same closure as the original call, `factTrace`: can be analyzed with a single template, just like `fact`.

Recursive customization happens when an unbounded number of templates are required to analyze some recursive method *M*. To create the demand for an unbounded number of *M*-templates, the analysis of each *M*-template must, directly or indirectly, lead to the creation of a new closure object type that, directly or indirectly, is used as an argument for *M*.

In contrast to the first necessary condition given above, this sharper condition does not translate into a simple programmatic test. Indeed, to base a test on it, one would first need to specify the precise meanings of "directly or indirectly" and "lead to." Since the question of non-termination is intricately tied with details of other parts of the type inferencer, we decided against basing tests on this sharper condition. Had we done it, the test would be fragile and possibly need repairs when these other parts of the type inferencer were modified. For example, an exact test for recursive customization could not ignore the strategy for dealing with megamorphic sends, nor could it ignore the rules for fusing closure object types:

- Megamorphism (see Section 3.3.2) can prevent recursive customization, since it causes the type inferencer to customize less aggressively.
- Fusing the closure object type in the recursive send with a closure object type from a previous send (see Section 4.4.2) breaks recursive customization, since both sends can be connected to the same template.

To avoid using a fragile test in an experimental system seeing constant change, our efforts have been directed towards devising conservative tests that are precise, yet do not sacrifice robustness to gain every bit of precision possible.

5.1.2 The Self virtual machine

Recursive customization not only affects type inference. The Self compilers [26, 58] are also susceptible to it because they customize methods, i.e., generate translated versions that are specialized for certain types of arguments†. It takes only a small program to demonstrate the problem:

```
traits block _AddSlots: ( |
  snail = ( '#' print. [value] snail. ).
| )

[0] snail.
```

Executing this program with an option set that causes the compiler to print the name of the methods it compiles results in this output:

```
Compiling snail
#Compiling snail
#Compiling snail
#Compiling snail
...
```

Each time `snail` is invoked, the receiver is a new closure, and the virtual machine recompiles `snail` to customize it to the new receiver. Recursive customization causes non-termination of type inference, but it affects the compiler less dramatically: programs still run, but slowed down by the compilation overhead.

In Self, recursive customization happens less often during compilation than type inference. The compilers are less susceptible because they normally customize on the receiver only, whereas the cartesian product algorithm customizes symmetrically on all arguments. Hence, recursive customization can only happen during compilation if a recur-

† Chambers [28] introduced the word "customization" to describe the recompilation of inherited methods for each kind of object inheriting them. Customization makes the type of `self` known in the customized methods and allows subsequent optimizations, such as static binding and inlining of sends to `self`.

sive send has a closure as the receiver. Most Self methods that execute with a closure as the receiver are found in `traits block`. These methods change infrequently, and are—by design or incidentally—not recursive. Thus, recursive customization is a rare curiosity during compilation and execution of Self programs. Still, if a programmer wants to write (the peculiar?) `cpsFact:` which is just like `factCPS:` except that the argument and receiver are swapped, the Self compilers unfortunately impose compilation overhead on each invocation of `cpsFact:`.

While recursive customization is a minor problem in the Self virtual machine, overcoming it without losing the performance benefits of customization may require going beyond the current approach of compiling a method at a time, because indirect recursion makes detection of recursive customization a non-local problem. (This will become clear in Section 5.3.) Inlining helps some in detecting recursive customization, since it allows callees to be compiled and analyzed as part of the caller. However, we doubt that inlining can always make detection of recursive customization possible within a single (expanded) method. Kaser, Ramakrishnan, and Pawagi discuss the general problem of converting indirect recursion to direct recursion by inlining [68].

5.1.3 Indirect recursion

So far, we have illustrated recursive customization with simple directly-recursive methods where the recursive calls are located in the same methods they invoke. For example, the `fact` method generates this call sequence:

$$\texttt{fact} \rightarrow \texttt{fact} \rightarrow \texttt{fact} \rightarrow \texttt{fact} \rightarrow \ldots$$

Recursive call sequences are often more complicated. Indeed, the “typical” recursive call is more complicated because it is found in a block[†], as illustrated by these two examples:

```
factIndirect = (
  0 = self ifTrue: [1] False: [self * (self - 1) factIndirect].
).

7 factIndirect printLine.

factIndirectCPS: resultBlk = (
  0 = self ifTrue: [resultBlk value: 1]
            False: [self-1 factIndirectCPS:
                          [|:a| resultBlk value: a*self].
                   ].
).

(8 factIndirectCPS: [|:a| a]) printLine.
```

Both `factIndirect` and `factIndirectCPS:` generate recursive cycles that are three calls long:

$$\texttt{factIndirect} \rightarrow \texttt{ifTrue:False:} \rightarrow \texttt{value} \rightarrow \texttt{factIndirect} \rightarrow \ldots$$

(analogous for `factIndirectCPS:`). Self’s approach of defining control structures using objects and messages means that functions that would be directly recursive in C or Pascal are often indirectly recursive in Self. The conditional statement in `factIndirect` would not produce method calls in the equivalent Pascal program, and hence would not intersperse

$$\ldots \rightarrow \texttt{ifTrue:False:} \rightarrow \texttt{value} \rightarrow \ldots$$

between each recursive invocation. This substitution of indirect recursion for the simpler direct recursion makes it harder to analyze recursive Self programs since detecting indirect recursion requires a global analysis, whereas direct recursion can be detected on a per-method basis.

[†] Most recursive calls are conditional, which in Self usually means nested in a block.

Like fact, factIndirect can be analyzed straightforwardly because the recursive call has the same receiver type, {$\overline{\text{smallInt}}$}, as the original call. Indeed, the inferencer need only create one factIndirect template. And like factCPS:, factIndirectCPS: causes recursive customization because each recursive call passes a new closure. However, it is harder to detect the recursive customization in factIndirectCPS: than in factCPS: due to the indirection. Figure 52 shows how the recursive customization in factIndirectCPS: involves several methods (compare with the simpler Figure 51 describing factCPS:). As Figure 52 illustrates, when the type inferencer creates a factIndirectCPS: template, it also creates two new closures, one for each of the blocks

```
[^ resultBlk value: 1]
[self-1 factIndirectCPS: [|:a| resultBlk value: a*self]].
```

These closures are passed to templates for the ifTrue:False: methods in true and false. The latter case invokes the second closure, yielding a new template for the block method

```
( self-1 factIndirectCPS: [|:a| resultBlk value: a*self] ).
```

The block method contains another block, [|:a| resultBlk value: a*self], from which a new closure is produced. Finally, this closure is passed to factIndirectCPS: by the recursive call, and the sequence repeats.

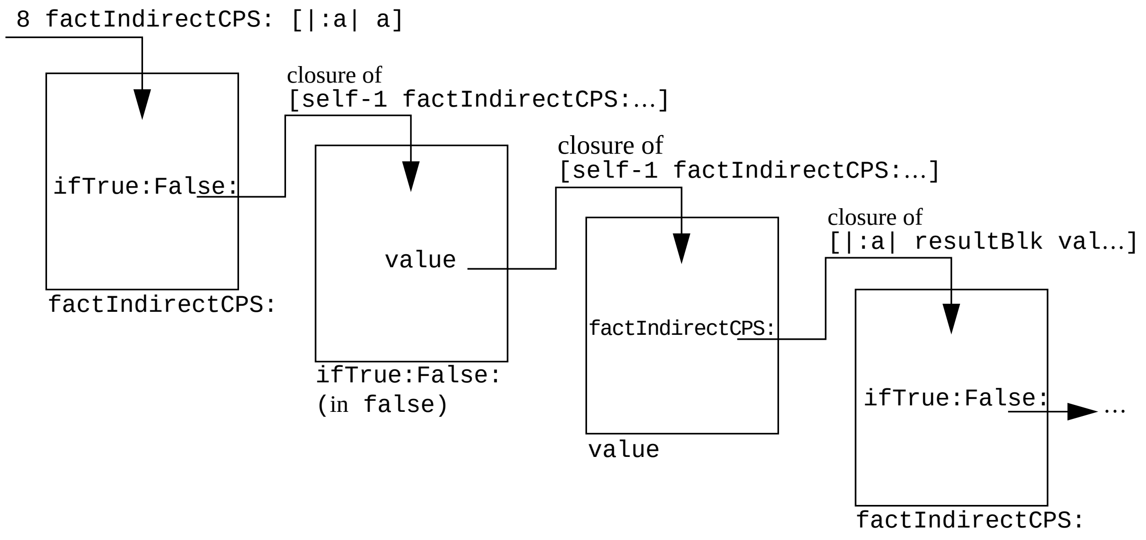


Figure 52. Recursive customization in factIndirectCPS: method

The indirect recursion in factIndirectCPS: does not pass closures of a single block through the full recursive call sequence. Instead, it passes a chain of closures, each one depending on the previous one. This pattern complicates detection of recursive customization over simpler directly recursive cases.

To determine that the analysis of factIndirectCPS: involves recursive customization, the type inferencer must recognize that the call sequence factIndirectCPS: → ifTrue:False: → value may be repeated, and that for each repetition, a new *chain* of closures may appear. Unlike the directly recursive case, no method is invoked on a closure of its own, and no single closure is passed through the full recursive call sequence.

As will become clear in Section 5.3, indirect recursion makes detection of recursive customization much harder. It is useful to distinguish several classes of recursion since the detection algorithms presented in Section 5.3 offer different coverage.

Definition. *Direct recursion* occurs when a method invokes itself. Note that the recursive send is not allowed to be in a nested block. Example: fact.

Definition. *Lexical indirect recursion* occurs when the recursive send is found in a block method nested in the method being invoked. (Normally the send invokes the *outer* method; it is unusual, although possible, for a block method to invoke an enclosing block method.) Example: `factIndirect`.

Definition. *Non-lexical indirect recursion* covers all remaining recursive methods. For example, two outer methods calling each other:

```
zip: b1 = ( self zap: [b1 value + 1] ).
zap: b2 = ( self zip: [b2 value + 2] ).
```

Definition. *Lexical recursion* denotes recursion that is direct or lexically indirect (viewing direct recursion as invoking the method at the lexical distance of zero from the method containing the recursive send).

Lexical recursion covers most recursive sends in practice. However, we have found two frequent cases in the Self world where non-lexical indirect recursion causes recursive customization:

- *Double-dispatching of three-way comparisons of bigInts and smallInts*. This code contains indirect recursion between the methods `compare:IfLess:Equal:Greater:` in `traits bigInt` and two `compareBigInteger:IfLess:Equal:Greater:` methods in `traits smallInt` and `traits bigInt`, respectively. Using subscripts to distinguish the latter two, and abbreviating the suffix "`IfLess:Equal:Greater:`" to "`L:E:G:`", the type inferencer finds that these methods may call each other as follows, yielding a recursive cycle of length 4:

  `compare:L:E:G:` → `compareBigInteger:L:E:G:`$_{\text{bigInt}}$ → `compare:L:E:G:` → `compareBig-Integer:L:E:G:`$_{\text{smallInt}}$ → `compare:L:E:G:` → …

- *Foreign function error recovery.* The invocation of a foreign function fails if its dynamic library has not been loaded. A failure handler then takes control, loads the library, revives the foreign function proxy, and retries the invocation. The failure handling code contains 23-way indirect recursion, as follows[†]:

  `lookupFunction:IfFail:` → `lookupFunction:ResultProxy:IfFail:` → `protect:` → `ifTrue:False:` → `value` → `ifFalse:` → `ifTrue:False:` → `value` → `loadIfFail:` → `incCountIfOne:` → `protect:` → `ifTrue:False:` → `value` → `do:` → `doLinks:` → `whileFalse:` → `whileTrue:` → `loop` → `value` → `value` → `value:` → `value:With:` → `value:` → `lookupFunction:IfFail:` → …

These examples complete our analysis of the recursive customization problem. Next we discuss how to handle the problem, structuring the presentation according to the two subproblems identified in the introduction: avoidance of recursive customization and detection of recursive customization. We start with avoidance of recursive customization, assuming that it has somehow been detected.

5.2 Avoiding recursive customization

Normally, the cartesian product algorithm customizes templates to a specific combination of arguments (i.e., the formal argument types are sets of size one). To be specific, assume that the type inferencer is analyzing a send S in the context of a template T_{current}. The core of the cartesian product algorithm consists of two nested loops. The outer loop iterates the variable M over the target methods of S, while the inner loop iterates the variable A over the cartesian product of the types of the arguments that may be passed to M. For simplicity, we ignore that both loops are implemented lazily to support the continuous arrival of new targets and/or actual argument object types. Instead we consider the loops abstractly, as follows:

[†] In Release 4.0 of the Self system, dynamically-linked libraries are used less than in Release 3.0 so this instance of indirect recursion, while still in the image, does not show up as frequently any more.

```
for all $M$ in "targets for send $S$ in template $T_{\text{current}}$" do
    for all $A$ in "cartesian product of arg. types for send $S$ in template $T_{\text{current}}$ invoking $M$" do
        "Connect $S$ to an $M$-template customized for $A$; if no such template exists, create it"
    end
end
```

To avoid recursive customization, less customization must be done in the critical cases; in the non-critical cases, nothing need be done differently. The first step in avoiding recursive customization is therefore to insert a test to distinguish the critical cases:

```
for all $M$ in "targets for send $S$ in template $T_{\text{current}}$" do
    for all $A$ in "cartesian product of arg. types for send $S$ in template $T_{\text{current}}$ invoking $M$" do
        if RecursiveCustomization($S$, $T_{\text{current}}$, $M$, $A$) = false then
            "Connect $S$ to an $M$-template customized for $A$; if no such template exists, create it"
        else
            "Customize less (than above) to ensure termination"
        end
    end
end
```

The test `RecursiveCustomization`(S, T_{current}, M, A) conservatively determines if recursive customization may be an issue. Section 5.3 discusses how to implement it. With this test in place, we need to devise a less aggressive customization strategy for the (occasional) cases when recursive customization has been detected. By customizing less, fewer new templates are required, and therefore fewer new closures are created, ensuring termination of type inference.

In the normal case, when recursive customization is not a concern, the cartesian product algorithm connects the send to an M-template customized for the exact combination of actual arguments in A. If no such template exists, the type inferencer creates one at this point. In the recursive customization case we modify this strategy. Instead of insisting on using a template customized *exactly* to A, the inferencer searches the existing M-templates for one that is a reasonably good, but perhaps not perfect, match.

Section 5.2.1 describes the search for a usable template and what to do if it fails. Section 5.2.2 describes a heuristic for picking the best template, should the search deliver several candidates.

5.2.1 Progressively widening search

Recall that the type inferencer is analyzing a recursively customizing send S in a template T_{current}, a target method M, and a tuple of actual argument types A. To analyze the send S, the inferencer must locate (or if necessary create) an M-template to connect S to. We propose, and have implemented, a strategy that searches for a suitable M-template in three progressively larger classes of M-templates. The search only progresses from one class to the next if no acceptable template has been found yet. In the order searched, the three classes are:

- *Class 1*. The set of existing M-templates that match A exactly.

  This set is either empty (if a template customized to A does not exist) or contains a single template, T_{exact}. In the latter case, T_{exact} is the same template as the cartesian product algorithm would have connected the send to, had recursive customization not been an issue. This case can be considered the fortunate case: there happened to be an exactly matching template lying around.

- *Class 2*. The set of M-templates found among the "nearby" ancestors of the template T_{current}.

  If a recursive call chain has started unrolling, the type inferencer will likely find a suitable template for the recursive send among the ancestors of the current template T_{current} (unless the recursive customization has just begun; typically, however, the detection algorithms need to let a few cycles unroll before they "see the pattern"). How far should the inferencer look back? The backtracking should proceed only to callers that are passing closures, i.e., it should not retreat past the point in the call graph where the recursive customization began. Limiting the backtracking in this manner prevents the introduction of “cross constraints” between different recursive calls and ensures better precision of type inference, as the following example illustrates.

The method `fact:IfOverflow:` computes the factorial of a smallInt, but if an overflow occurs, returns the value obtained by invoking an overflow block (in the example `multiply:IfOverflow:` denotes an overflow-checking multiplication operation):

```
fact: n IfOverflow: blk = (
  0 = n ifTrue: [1]
         False: [(fact: n - 1 IfOverflow: [^ blk value])
                  multiply: n IfOverflow: blk].
).
```

Assume that a program contains two sends that invoke this factorial method:

```
fact: 20 IfOverflow: [nil].
fact: 17 IfOverflow: ['overflow'].
```

To analyze the two sends, the inferencer connects them to different templates, since they supply different overflow blocks. Each of these `fact:IfOverflow:` templates will then grow their respective recursive call graphs; see Figure 53. At some point, the type inferencer detects the recursive customization and attempts to break it. If breaking the recursion involves connecting the recursive sends to ancestors in their own call graphs, the result types from `[nil]` and `['overflow']` do not mix, and the types of the two sends originally invoking `fact:IfOverflow:` are inferred precisely as $\{\overline{\text{smallInt}}, \overline{\text{nil}}\}$ and $\{\overline{\text{smallInt}}, \overline{\text{string}}\}$. On the other hand, if breaking the recursion connects a recursive send in one of the call graphs to a template in the other call graph (creates a “cross constraint”), the contributions from the two overflow blocks can no longer be kept apart, and the imprecise type $\{\overline{\text{smallInt}}, \overline{\text{string}}, \overline{\text{nil}}\}$ is inferred for both sends.

- *Class 3*. The set of all existing *M*-templates.

  Clearly, termination can be ensured without the class 3 search, but at the expense of occasionally unrolling an extra recursive call. Thus, in the implemented type inferencer, we have left it optional whether to perform the class 3 search. On one hand, allowing it may result in faster type inference because fewer templates are created. On the other hand, as illustrated by the `fact:IfOverflow:` example, precision may be lost because class 3 searches can introduce cross edges.

If none of the three classes contain an acceptable template for the given send and argument tuple, according to the criteria given in the next section, the type inferencer creates a new template and connects the send to it. The inferencer marks the new template as “less customized,” to ensure that the next time through the recursive customization cycle, it will be accepted during the class 2 search.

5.2.2 Selecting a template

Given a template $T_{\text{candidate}}$, the type inferencer needs to test whether it is acceptable to connect a recursive send S with a tuple of arguments A to $T_{\text{candidate}}$. Furthermore, if several templates are acceptable, the inferencer needs to identify the best one. Our type inferencer computes a numerical measure, a “penalty,” for each template to solve both of these problems. If a template’s penalty is smaller than a fixed threshold, the template is deemed acceptable; when several templates are acceptable, the inferencer picks the one with the lowest penalty (breaking ties arbitrarily). Intuitively, the penalty assesses the pollution of the template’s formal argument types that would result from connecting the send to the template: the more new object types that would be propagated into the formal arguments of the template, the higher the penalty.

Before we describe how to compute the penalty, we need to explain an aspect of the implementation of the type inferencer. In the implemented system, type variables come in two kinds: frozen and non-frozen. *Frozen* type variables are created with a specific value (type) and subsequently never change. *Non-frozen* type variables are created with an initial value (usually the empty set) and may grow monotonically as type inference progresses. Most of the time the cartesian product algorithm can use frozen type variables of size one for formal arguments because it customizes

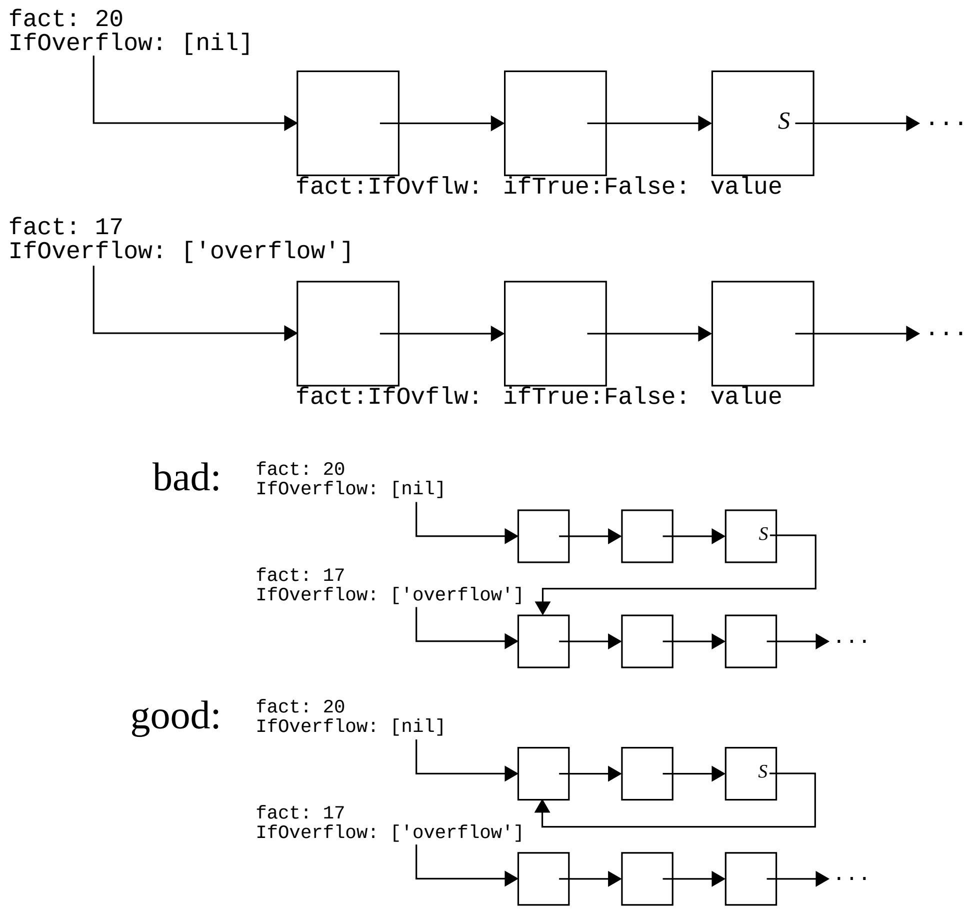


Figure 53. Two ways to break recursive customization

We restrict the class 2 search for an acceptable template to the ancestors of the template containing the recursive send *S*. Without this restriction, "cross constraints" between different recursive calls chains can cause precision loss. For instance, the "bad:" illustration shows how `nil` and `'overflow'` mix. Imposing the restriction favors the connection labelled "good:," and therefore avoids the precision loss.

templates to specific tuples of arguments[†]. Frozen type variables can also be used for many primitive sends, for constant slots, and in general whenever a type variable has a fixed number of incoming constraints that all originate in frozen type variables. The fact that a type variable is frozen carries important information:

- The representation of the type variable can be optimized since its size and exact value is known immediately. For example, the Self type inferencer canonicalizes all frozen type variables of size one (a large fraction of all type variables).
- The type inferencer can reason precisely about expressions that have frozen type variables; instead of merely knowing lower bounds (subsets) of the type, the exact type is known. For example, the inferencer exploits this property when fusing closure object types; see Section 4.4.2.

We now describe how to compute the penalty for a template $T_{\text{candidate}}$. For $i = 0, 1, \ldots, k$, let F_i denote the type variable of the i'th formal argument in $T_{\text{candidate}}$, and let A_i denote the type variable of the i'th actual argument in the argument tuple A. The penalty is the sum of argument-specific penalties over all arguments:

[†] There are two exceptions when the cartesian product algorithm may elect not to customize an argument to a type of size one: megamorphic actual arguments and ignored formal arguments (see Section 3.3.2).

$$\text{penalty}(T_{\text{candidate}}) = \sum_{i=0}^{k} \text{penalty\_arg}(T_{\text{candidate}}, A_i, F_i) .$$

The penalty for the *i*'th argument assesses how badly the constraint

$$\begin{tikzcd} \bigcirc_{A_i} \arrow[r] & \bigcirc_{F_i} \end{tikzcd}$$

would pollute F_i. The penalty is computed by a case analysis. The first case distinguishes between arguments that are ignored by the invoked method (represented by $T_{\text{candidate}}$) and those that are not:

$$\text{penalty\_arg}(T_{\text{candidate}}, A_i, F_i) = \begin{cases} 0 & \text{if } T_{\text{candidate}} \text{ ignores } F_i \\ \text{penalty\_arg}(A_i, F_i) & \text{otherwise.} \end{cases}$$

Since ignored arguments cannot affect the precision of type inference, there is no penalty for polluting them. The penalty for a non-ignored argument is computed by a case analysis on the *current* values of A_i and F_i, taking into account whether or not they are frozen. Table 21 gives the details. In the table, the following symbols are used:

- c_i: the number of new *sibling* closure object types that the constraint $A_i \subseteq F_i$ would immediately propagate into F_i. A sibling closure object type is a closure object type for a block that already has one or more closures in F_i.
- r_i: the number of new non-sibling object types that the constraint $A_i \subseteq F_i$ would immediately propagate into F_i.

Thus, each object type added to F_i is counted in either c_i or r_i.

The expressions in Table 21 capture the following intuition. In the first column where F_i is frozen, the possible penalties are extreme: 0 or ∞. The penalty is 0 when A_i is and will remain a subset of F_i (thus, no pollution of F_i occurs if the send S is connected to $T_{\text{candidate}}$). Otherwise the penalty is ∞, reflecting that it is a hard error to add new object types to the frozen type variable F_i (the infinite penalty immediately makes $T_{\text{candidate}}$ unacceptable for S, preventing the error from happening). In the second column where F_i is not frozen, each new sibling closure object type incurs a penalty of 1, whereas other new object types carry a penalty of 10 each. Moreover, if A_i is not frozen, an additional penalty of 10 is incurred (this fixed penalty approximates the damage from potential future members of A_i).

	F_i frozen	**F_i not frozen**
A_i frozen	0 if $A_i \subseteq F_i$; ∞ otherwise	$10r_i + c_i$
A_i not frozen	∞	$10 + 10r_i + c_i$

Table 21. How penalty_arg(A_i, F_i) is determined

The penalty assessment is based on the assumption that the current values of A_i and F_i are fairly good approximations of their final values. We have found this assumption to hold in practice, but it should be noted that recursive customization occurs infrequently enough so that minor inaccuracies when resolving it are unlikely to affect overall precision. Should the outlined approach fall short, it can be modified independently of the code that *detects* recursive customization. One particular alternative worth mentioning is the use of iteration. By iterating, the previous iteration's values for the type variables A_i and F_i can be used instead of the current values (while types inferred in the previous iteration are also approximations, they might be better approximations than the current types of the present iteration).

Once the type inferencer has computed the penalties, it is straightforward to process a recursive send. For each class of templates searched, the inferencer ranks the templates according to their penalty and connects the send to the one with the lowest penalty—assuming that the penalty is less than the threshold for the given class. In the implemented type inferencer, the threshold defaults to infinity, i.e., any template with a finite penalty is acceptable. We do not claim that this extreme threshold is optimal, although it seems to work well in practice. The opposite extreme is to never accept a template that has an argument-specific penalty of strictly more than 1 (the lowest value that can still ensure termination, since merging sibling object types—a merge we must accept to guarantee termination—incurs a penalty of 1).

Our approach to managing recursive customization involves several numerical parameters: the thresholds, the weights multiplied on r_i and c_i, and the penalty for non-frozen actual arguments. The parameters could conceivably be adjusted to ensure faster or more precise type inference, although overall improvements would likely be minor since situations where the exact values of the parameters make a difference are quite rare (given that recursive customization is rare in the first place). The current settings of these parameters, while probably not optimal, represent our best initial guesses, but we have not found dramatic effects by adjusting them within reasonable bounds. With these remarks, we turn our attention from handling recursive customization to detecting it.

5.3 Detecting recursive customization

The monotonicity requirement of type inference means that recursive customization should be detected before connecting the critical send to a template (it is hard to disconnect the send, should recursive customization be detected after the fact). However, the situation is more forgiving, since dealing with recursive customization can always be deferred until the next recursive cycle, although at the expense of increased type inference time. In the previous section. we met the early detection requirement by inserting a test for recursive customization just prior to the code connecting the send. In this section, we describe how to implement the test, i.e., how to implement this procedure:

```
Procedure RecursiveCustomization($S$, $T_{current}$, $M$, $A$)
  (* $S$ is a send in the template $T_{current}$, $M$ is a target method, and
     $A$ is a tuple of actual arguments. Return true if recursive
     customization may be happening. *)
...
end RecursiveCustomization.
```

The test for recursive customization is in the critical loop of type inference. It is invoked every time the inferencer processes a send (at least in principle). Thus, the test must be fast to avoid an overall slowdown of inference. More precisely, since the majority of sends do not cause recursive customization, this case should be optimized. We want a test that rules out recursive customization quickly most of the time. Then, if necessary, we can then afford to spend more time on the few remaining sends.

Although the task is detection of recursive customization, detecting recursion is a natural first step. In related work [93, 121] other researchers have partitioned the call graph into strongly connected components (SCCs) to detect recursion (a SCC is a subgraph in which there is a path from any node to any other node). The methods in each SCC are (or may be) mutually recursive. In the present case, this approach does not apply: at the source level we have no control-flow information, and at the template level we are trying to *introduce* a cycle to prevent run-away unfolding: until we introduce the cycle the recursively customizing methods do not correspond to a SCC.

We present three approaches for timely detecting recursive customization. The approaches offer different compromises between:

- *Coverage*: how many cases of recursive customization the test detects.
- *Performance*: how much CPU time the test consumes.
- *Convenience*: the degree of programmer involvement required.

Section 5.3.1 describes an approach based on programmer-inserted declarations. Section 5.3.2 describes an automatic detection algorithm that is fast, but limited to recognizing recursive customization driven by lexical recursion. Section 5.3.3 presents a more powerful automatic detection algorithm, capable of recognizing all cases of recursive customization. Finally, Section 5.3.4 compares and summarizes the different approaches for detecting recursive customization.

5.3.1 Programmer declarations

The simplest approach, and the one we initially implemented, requires the programmer to add declarations to sends that may cause recursive customization. For example, to avoid the recursive customization in the 23-way indirectly recursive case mentioned in Section 5.1.3, a declaration could be added to the `lookupFunction:ResultProxy:IfFail:` send in the `lookupFunction:IfFail:` method. Similarly, the programmer should declare the recursive send in `factCPS:` as recursively customizing, but can omit declarations from the recursive call in `factIndirect:`, knowing that it does not cause recursive customization.

Using programmer declarations for detecting recursive customization has two principal advantages:

- *High performance*. Declarations incur virtually no testing overhead.
- *High precision*. The programmer can sometimes do a better (i.e., less conservative job) than the automatic algorithms. However, to realize high precision, the declaration language may have to be quite sophisticated, e.g., allowing the programmer to express conditional properties like “if the receiver has a certain type and the argument is a closure with scope in a specific method, recursive customization may happen.”

The disadvantages of using declarations are the usual ones associated with redundant and unchecked specifications. Programmers may forget to add declarations or forget to update them when they change methods. For instance, if the `lookupFunction:ResultProxy:IfFail:` is changed in a way that removes the indirect recursion, the programmer must remember to remove the declaration in `lookupFunction:IfFail:`.

The consequence of extra declarations may be precision loss. The consequence of missing declarations may be non-termination. Given some expectations for how long type inference should take, the latter problem can be dealt with reasonably well. The idea is to run type inference for a bit longer than the expected time to completion. If it terminates, everything is fine. If it diverges, recursive customization must be suspected. The computation can then be stopped, and missing declarations should be located. A simple tool that finds the template T_{distant} with the highest distance from the main template and prints the path from the main template to T_{distant}, can often pinpoint missing declarations. To illustrate, we inferred types with no declarations and no automatic detection for this method (see Section 5.1.3 for the definition of `factIndirectCPS:`):

```
main = ( (8 factIndirectCPS: [|:r| r]) printLine. ).
```

After a few minutes of waiting in vain for the result, we stopped the inferencer and invoked the path tracing tool. The output, shown in Figure 54, clearly indicates a missing declaration in `factIndirectCPS:`.

Programmer declarations may be a good option if very few are required: the inconvenience is minor, the chance of errors is low, and the performance penalty of a declaration-less approach is hard to justify.

5.3.2 Lexical search

Lexical recursion can be detected by traversing the lexical chain of the calling method and testing if any of the enclosing methods equal the invoked method:

```
Procedure RecursiveCustomization($S$, $T_{\text{current}}$, $M$, $A$)
begin
    if “the target method $M$ does not lexically enclose the calling method $T_{\text{current}}$.method” then
        return false
    end;
    return true
end RecursiveCustomization.
```

Distance	Selector
688	factIndirectCPS:
687	value
686	ifTrue:False:
685	factIndirectCPS:
...	
7	factIndirectCPS:
6	value
5	ifTrue:False:
4	factIndirectCPS:
3	value
2	ifTrue:False:
1	factIndirectCPS:
0	main.

Figure 54. Recursive customization revealed by backtracking from the most distant template

Typical Self methods are not nested deeply, ensuring a fast test of whether or not the send in question is lexically recursive. There are, however, two problems with this test:

- *Lack of coverage.* The test does not detect non-lexical indirect recursion, and thus cannot stand alone.
- *False positives.* The test is overly sensitive. It detects lexical recursion, but as seen in Section 5.1.1, a method can be recursive without causing recursive customization.

The first problem can be addressed by supplementing the lexical test with declarations, a possibility we briefly discuss in Section 5.3.4. The second problem can be addressed by refining the test. In Section 5.1.1, we observed that recursive customization can happen only if the recursive send passes a closure. Thus, we can sharpen the lexical test and suppress many false positives by augmenting it with an inspection of the arguments:

```
Procedure RecursiveCustomization($S$, $T_{\text{current}}$, $M$, $A$)
begin
    if “the target method $M$ does not lexically enclose the calling method $T_{\text{current}}$.method” then
        return false
    end;
    if “all arguments in $A$ are frozen and contain no closure object type” then
        return false
    end;
    return true
end RecursiveCustomization.
```

We treat non-frozen argument types conservatively, assuming that they may later contain a closure object type. With other type-inference algorithms than the cartesian product algorithm, fewer arguments would be frozen, and the refinement of the test less effective.

With the refined test, false positives may still occur. For example, `factTrace:` from Section 5.1.1 demonstrates that even if a recursive call passes a block closure, the closure may not be “renewed” between each recursive call, and therefore cannot cause non-termination of type inference. Nevertheless, for typical Self programs, the fraction of sends testing positive is quite low, as shown in Table 22. Thus, the overall precision gain resulting from additional elimination of false positives would likely be minor.

	Polyvariant counting (over templates, not syntactically)	
	# of sends	**# of sends testing positive**
HelloWorld	19	0
Factorial[a]	29,641	0
Richards	32,484	0
DeltaBlue	44,118	0
Diff	62,112	0
PrimMaker	41,596	0
Fact	29,667	0
FactCPS	29,717	8
FactIndirect	29,661	0
FactIndirectCPS	29,703	8

Table 22. The number of sends testing positive under the lexical test

a. The Factorial program is very similar to FactIndirect. We include it here for consistency with measurements given in the rest of this dissertation.

5.3.3 Call-graph search

To detect all cases of recursive customization, including those driven by non-lexical indirect recursion, a lexical search is insufficient, since there may not be a lexical relationship between the methods involved in the recursion. Instead, it is necessary to search for repeated patterns in the call graph that the type inferencer builds. This has two immediate consequences, both of which increase the computational cost of testing:

- The call graph depth usually is not limited by a small constant, unlike the lexical nesting depth of blocks.
- The call-graph search, unlike the lexical search, is not confined to simple linear chains. Rather, in the call graph, each template may have several incoming and outgoing edges and cycles may be present.

Below, we outline an algorithm that, based on a call-graph search, can detect any occurrence of recursive customization. First, Section 5.3.3.1 describes a simple version of the algorithm. Like the simple lexical search, the simple call-graph search conservatively detects recursion, not the more specific recursive customization. Thus, it suffers from false positives. Section 5.3.3.2 describes several ways to improve the precision and efficiency of the call-graph search algorithm.

5.3.3.1 Simple call-graph search

During analysis, the type inferencer gradually builds a call-graph approximation. The nodes in the call graph are templates. The edges in the call graph reflect possible method invocations: the call graph has an edge from T_1 to T_2 if T_1 contains a send expression that is connected to T_2. (Note: the edges in the call graph are *not* constraints; constraints reflect data flow, whereas the edges in the call graph reflect control flow.) Figure 55 illustrates such a call graph. The figure omits return edges, since these are not important for the detection of recursive customization.

The call graph can be used to detect recursive customization or, as a first cut, detect recursion. The idea is to look for call sequences (paths) in the call graph that repeat methods, e.g.:

$$\texttt{main} \to \texttt{factIndirect} \to \texttt{ifTrue:False:} \to \texttt{value} \to \texttt{factIndirect} \to \dots$$

Such paths correspond directly to possible stack configurations during execution, e.g., the above path reflects that during execution of the program, the stack may contain (from bottom to top) activation records for these methods: `main`, `factIndirect`, `ifTrue:False:`, `value`, `factIndirect`, etc.

Consider the situation in Figure 55. A template T_{current} contains a send *S* that invokes a method *M* with actual arguments *A*. To determine whether this situation may lead to recursive customization, the call-graph search algorithm considers all simple (i.e., acyclic) paths from the main template to T_{current}. If some path, like the one shown with bold arrows on the figure, already contains an *M*-template, this indicates that *M* may be recursive. Consequently, the call-graph search algorithm conservatively declares that recursive customization may happen. On the other hand, if no path contains an *M*-template, the call-graph search algorithm can safely declare that the send does not cause recursive customization.

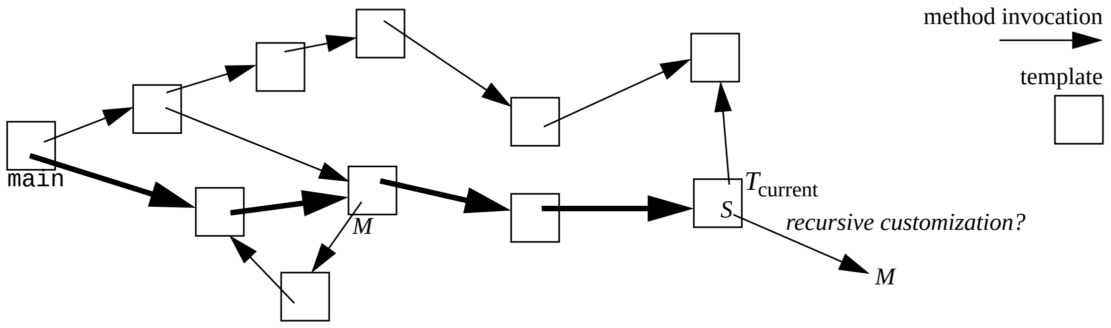


Figure 55. Call-graph search to detect ongoing recursive customization

To determine if a send *S* invoking a method *M* may cause recursive customization, all simple paths from the main node to the template containing *S* are enumerated. If one or more paths contain an *M*-template, like the bold path above, the send *may* cause recursive customization.

While the simple call-graph search algorithm is safe in the sense that it never overlooks an occurrence of recursive customization, it has two problems: it is imprecise because it warns about recursive customization in many cases where it does not actually occur, and it can be slow. Both problems must be addressed to make the call-graph search an effective technique. The next section describes improvements to alleviate these problems.

5.3.3.2 Improved call-graph search

Parameterizing the sensitivity. The call-graph search lacks precision because it “jumps to conclusions” immediately upon encountering a path containing the target method *M*. To avoid false positives, the sensitivity must be lowered. To this end, we introduce an integer parameter *L* (for “levels”), and modify the search algorithm to declare that recursive customization may happen only after finding a simple path with more than *L* templates. For example, the situation in Figure 55 would not result in a positive test if $L > 1$.

As *L* is increased, the call-graph search gets less sensitive and yields fewer false positives. The price, however, is delayed detection of recursive customization when it does happen: a larger number of recursive calls must be unrolled before a sufficiently repetitive path occurs. Any delay slows down type inference because more templates are created. For this reason, only moderately high values of *L* work well. The subsequent improvements acknowledge this problem and deliver more precision for a fixed value of *L*.

Testing argument types. Even if templates for some method occur repeatedly on a path, recursive customization may not be happening. Consider the nested conditional statements in Figure 56. (The implementation of `case` statements in Self nests `ifTrue:False:` statements even deeper, so this example is not unrealistic). Assuming that all conditions (`a=w`, `a=x`, `a=y`, `a=z`) are false, the active calls at the point (*) are:

$\texttt{ifTrue:False:} \rightarrow \texttt{value}_1 \rightarrow \texttt{ifTrue:False:} \rightarrow \texttt{value}_2 \rightarrow \texttt{ifTrue:False:} \rightarrow \texttt{value}_3 \rightarrow$
$\texttt{ifTrue:False:} \rightarrow \texttt{value}_4$

(We have indexed the `value` methods to clarify to which blocks they belong). A corresponding path with four `ifTrue:False:` templates will exist in the type inferencer’s call graph, so unless $L \geq 5$, the call-graph search algorithm *unnecessarily* declares that the last `ifTrue:False:` send causes recursive customization.

```
a = w ifTrue: [
    'w' print.
] False: [ "#1"
    a = x ifTrue: [
        'x' print.
    ] False: [ "#2"
        a = y ifTrue: [
            'y' print.
        ] False: [ "#3"
            a = z ifTrue: [
                'z' print.
            ] False: [ "#4"
                '?' print. "(*)"
            ].
        ].
    ].
].
```

Figure 56. Nested `ifTrue:False:` statements should not be considered recursive

While we could set *L* to 5 or more and avoid the imprecision in the specific example, this solution is unattractive for the reasons given above. Instead of increasing *L*, a more precise test can be obtained by involving the types of the arguments in the test for recursive customization. In the example above, each `ifTrue:False:` send passes closures of a different pair of blocks and therefore cannot cause recursive customization, since the program contains only finitely many different blocks. Thus, the following refinement of the call-graph search is safe:

- When counting *M*-templates on a path, count only those whose formal arguments have types "sufficiently similar" to those found in the actual argument vector, *A*, of the send being tested.

  More precisely, the *i*'th actual argument type A_i and the *i*'th formal argument type F_i of some template are similar if they have a non-empty intersection, or they contain a closure of the same block (the former condition takes care of all non-closure object types). Occasionally, A_i or F_i is not frozen, in which case we conservatively declare them similar.

With this refinement, the call-graph search counts only selected *M*-templates when processing a path. Thus, the counts will be lower, and for a given value of *L*, fewer sends test positive. In particular, the count for the last `ifTrue:False:` send in the nested conditional statements will be 0, so the send will not be deemed recursively customizing, even if the lowest possible value $L = 1$ is used.

Path pruning. The call-graph search can be slow because it searches long paths. However, it is both unnecessary and imprecise to search the full paths from the main template to T_{current}. Instead, only a suffix of each path should be searched. In Section 5.1.1, we observed that if some send does not pass a closure, this send cannot cause recursive customization. Thus, given a path, T_{main}, T_1, …, T_n, T_{current}, the type inferencer can search it backwards, counting *M*-templates only as long as each call on the path passes a closure object type in some argument position (again, when an argument type is not frozen the inferencer conservatively assumes that the argument may pass a closure). This way, instead of counting *M*-templates over the full path, the inferencer counts over a suffix only: T_m, T_{m+1}, …, T_{current}. Not only does the counting go faster, the counts will also be lower, and the test therefore more precise.

Avoiding path enumeration. Before the call-graph search algorithm can declare that a send does not cause recursive customization, it must (in principle) inspect all paths to determine that none of them have a particular property. Given that most sends are not recursive, this requirement is unfortunate: it makes the common case the most expensive one. To improve on this situation, we would like to avoid enumerating all paths.

One possibility may be to apply a variation of a data-flow analysis algorithm. Such algorithms compute "meet over all paths" solutions to data flow problems without enumerating all paths [70]. However, even if we can encode the call-graph search in a set of data flow equations and also incorporate the *testing argument types* and *path pruning* improvements, the resulting algorithm may still be less efficient than desirable.

Instead, we have designed and implemented *approximate counting* as a way to achieve efficiency directly. Rather than computing the maximum number of M-templates over all simple paths (subject to the *testing argument types* and *path pruning* optimizations), the type inferencer computes an upper bound. Comparing an upper bound against L may occasionally cause a send to be declared recursively customizing when it would not have been with an exact count, but we can always counterbalance this problem by increasing the level L. An upper bound can be computed by traversing the call graph backwards, starting in T_{current}. In each node, the type inferencer stores the maximum number of M-templates found thus far on any path from the node to T_{current} and the path that achieved this count (the path can be represented by a backpointer in each node). When backtracking along a call edge, the inferencer compares the current count against the one stored in the node. If the current count is greater and the node is not already on the path (we want to count over simple paths only), the inferencer has found a path to the node with more M-templates; it updates the count and the path in the node accordingly and proceeds backtracking. If the current count is smaller, no further backtracking along that edge is useful.

This backtracking computation yields an upper bound because it computes the maximum number of M-templates on simple paths ending at T_{current}; it does not verify that the paths can be extended back to the main template, i.e., it may include too many paths.

It is straightforward to add the *testing argument types* improvement to the backtracking counting; we count only the M-templates that satisfy the similarity condition on the arguments. The *path pruning* optimization is also easy to add; we restrict the backtracking to call edges that pass closure object types. We can think of this improvement as back-tracking in a sparser call graph where all edges that do not carry a closure argument have been deleted. Since the sparser graph has fewer edges, fewer templates will be reached, and the search will be faster and less conservative.

Sparse testing. Suppose recursive customization is happening and the recursive call sequence is:

$$M_1 \to M_2 \to M_3 \to M_1 \to M_2 \to M_3 \to \ldots$$

To break it, the type inferencer must fold *some* recursive call. It does not matter whether it picks $M_1 \to M_2$, $M_2 \to M_3$, or $M_3 \to M_1$: as long as it creates a cycle somewhere in the call graph, no more templates for the methods M_1, M_2, and M_3 will be needed, and the analysis terminates. This observation can be used to avoid testing for recursive customization at every call. In our implementation, the type inferencer performs the recursive customization test only when analyzing sends that are found in methods containing blocks. Somewhere on a recursive cycle such a method *must* exist to keep fueling the recursive customization with new closures object types (see Section 5.1.1). Since this sparse testing strategy ensures that the test will be applied at least once during every cycle, recursive customization will be detected within the same recursive call sequence as if the test had been applied at every send. The effect of avoiding some testing is limited for the fast lexical- and declaration-based tests, but for the slower call-graph search, it can make a noticeable difference on overall performance of type inference. In principle, the call-graph search test could be applied even less frequently, but the result may be later detection of recursive customization and an overall slow-down of type inference if the increased number of templates created more than offsets the savings from the reduced testing.

Along the same lines, the very first thing tested when looking for recursive customization is the actual argument vector A. If A does not contain a closure and never will (because all type variables in it are frozen), we know immediately that the call cannot cause recursive customization. Moreover, it suffices to perform the test for recursive customization when the type inferencer is about to create a new template since if a matching template already exists, the send cannot introduce the possibility of recursive customization.

To conclude the presentation of the recursive customization test based on the call-graph search, Table 23 gives statistics for how frequently the test comes out positive for a number of Self programs. In addition to the sends found positive under the call-graph search, some recursive sends are caught by the lexical test, which in our implementation always precedes the call-graph search (although the call-graph search provides complete coverage, the lexical test can often detect lexical recursive customization sooner than the call-graph search and therefore helps speed up type inference by folding recursive customization sooner). As the data in the table shows, few sends test positive under the call-graph search: recursive customization driven by non-lexical indirect recursion is simply not very frequent in these benchmarks, and the final version of the call-graph search is sufficiently precise to avoid false alerts.

	Polyvariant counting (over templates, not syntactically)	
	# of sends	**# of sends testing positive**
HelloWorld	19	0
Factorial	29,641	2
Richards	32,484	2
DeltaBlue	44,118	2
Diff	62,112	2
PrimMaker	41,596	2
Fact	29,667	2
FactCPS	29,717	2
FactIndirect	29,661	2
FactIndirectCPS	29,703	2

Table 23. The number of sends testing positive under the call-graph search test

5.3.4 Comparison of detection algorithms

We have described three different approaches to detecting recursive customization: programmer declarations, lexical search, and call-graph search. Table 24 summarizes the most important properties of each approach. The table also shows a fourth approach: combining programmer declarations and the lexical search. The combined test can automatically detect recursive customization when it is driven by lexical recursion, but needs declarations for the occasional non-lexical cases.† In conclusion, since the tests offer different trade-offs between coverage, computational cost, and programmer involvement, no single test is universally best.

	Coverage	**Convenience**	**Speed**
Declarations	all recursion	low	fast
Lexical search	lex. recursion	high	fast
Call-graph search	all recursion	high	slower
Decl's + lex search	all recursion	medium	fast

Table 24. Summary of the algorithms for detection of recursive customization

5.4 Summary

Recursion can interact with customization and cause non-termination of type inference. The interaction, termed recursive customization, happens when the analysis of a recursive send necessitates creation of a new template with another instance of the recursive send, and so on. Recursive customization affects not only adaptive type inference but also optimizing compilation based on generation of customized methods.

† The performance data published in [3] for the cartesian product algorithm were obtained using the combined test. Therefore, the data in [3] differ slightly from the data in Section 3.4 of this dissertation, where the call-graph search was used to prevent recursive customization.

We attacked the recursive customization problem by dividing it into two subproblems: detection and avoidance. Indirect recursion complicates detection by making it a non-local problem. We presented three basic approaches for detecting recursive customization: programmer declarations, lexical searching, and call-graph searching. The approaches offer different trade-offs between precision, efficiency, and convenience. Once recursive customization has been detected, it must be avoided by introducing a cycle in the call graph. To create the cycle, the type inferencer must customize less aggressively. We devised a customization strategy that searches three progressively larger classes of existing templates for a template to which the recursively customizing send can be connected. The templates within each class are ranked according to how closely their formal argument types match the actual argument types of the send in question, and the best matching template is chosen.

With these mechanisms in place, the Self type inferencer can analyze recursive methods without risking non-termination. Conceivably, the same mechanisms could be used to prevent recursive customization during compilation, although a non-local analysis would be required—something which the current Self compiler specifically avoids to maintain responsiveness.

6 Sifting out the gold: using types to winnow objects[†]

Integrated, dynamically-typed object-oriented programming environments like Smalltalk and Self boost programmer productivity through expressiveness, easy access to reusable objects, incremental program development, and fast turn-around. Nevertheless, their acceptance and use for production programming has been limited for two pragmatic reasons: execution speed lags an order of magnitude behind C's, and it is hard to deliver compact applications. Recently, great strides have been made towards leveling the playing field for performance of dynamically-typed object-oriented languages [26, 41, 59]. Application size, however, has remained an obstacle preventing programmers from using dynamically-typed exploratory programming environments. Without some sort of application extraction, even the simplest program such as HelloWorld has the same size as the entire programming environment.

This chapter reports on applying type inference to the task of extracting applications from integrated dynamically-typed environments. We have designed, implemented, and tested a new extraction algorithm that is

- *automatic*: it runs without programmer intervention, although it will accept advice to improve the results;
- *sound*: it preserves the full semantics of the extracted application;
- *efficient*: extraction of medium-sized applications takes seconds or a few minutes;
- *effective*: extracted applications shrink by an order of magnitude (of course depending on how big the application is in the first place; we describe a specific medium-sized example in detail below);
- *general*: although our implementation is specific to Self, the algorithm can be readily applied to, say, Smalltalk-80 and CLOS.

The rest of this chapter is organized as follows. Section 6.1 describes the extraction algorithm in detail. Section 6.2 presents measurements from extracting several test programs. Section 6.3 discusses issues raised by extraction; some related to the *design* of the extraction algorithm, and some related to the *use* of it. Section 6.4 reviews related work in a broader context than type-inference-based extraction. Finally, Section 6.5 summarizes and concludes.

6.1 How application extraction works

The input to the extractor is an image of objects containing an application to be extracted. Typically, the image is the standard Self image consisting of core objects, glue, user interface, and additionally one or more applications. The standard image of Release 4.0 comprises approximately 10 Mb. An *application*, as described in Section 3.1.1, is defined by a main object and main method. Like the `main()` function in a C program, the main object and method explicitly define how to start an execution of the application. In addition, they implicitly delineate the application: certain objects must be present for the execution to succeed, whereas other objects can be safely omitted. The extractor recovers this implicit delineation.

Our extraction algorithm consists of five steps, as shown in Figure 57. The first four steps, which include grouping and type inference, determine what needs to be extracted. The fifth step simply harvests the fruit and writes a source representation of the selected objects. The following sections use a running example, PrimMaker, to describe each step.

While detailed knowledge of PrimMaker is not needed, an overview is helpful. PrimMaker is a preprocessor that generates Self and C "glue code" for foreign functions; see Figure 58[‡]. For example, suppose a Self programmer needs to call the C function `open`. The programmer would create a "template file" containing a line naming the function and specifying the C types of its arguments and result. The programmer would next run PrimMaker on the file in order to generate three files. The first file contains a Self "wrapper" routine that calls a primitive, in this case named

[†] The material in this chapter was originally presented in [7].

[‡] This chapter refers to the January 1994 version of PrimMaker. The Release 4.0 version manipulates *annotations* [118], a reflective concept, which the type inferencer currently does not support.

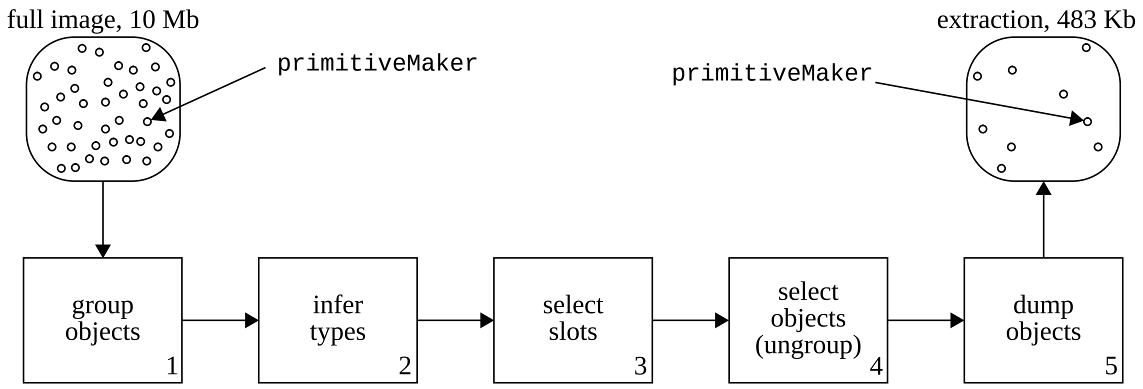


Figure 57. Overview of extraction

The extractor takes five steps to extract an application.

`_OpenName:Flags:Mode:`. The second file creates an entry in the Self virtual machine's primitive dispatch table, effectively defining the `_OpenName:Flags:Mode:` primitive. The third file contains a C++ stub that implements the primitive. The stub translates Self objects to C values, calls the `open` routine, and translates the return value back to a Self object. Many of the 600 primitives in the Self system are implemented in this manner, using PrimMaker.

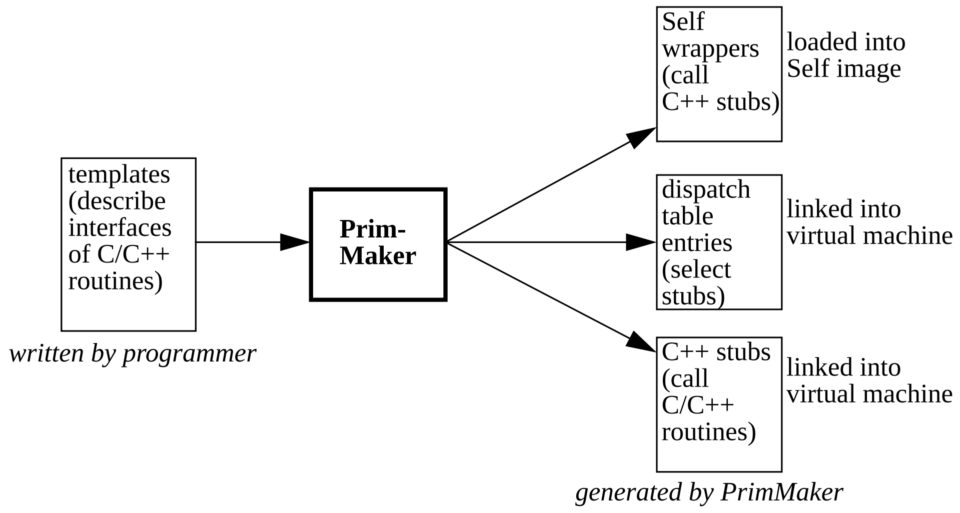

templates (describe interfaces of C/C++ routines)
written by programmer
Prim-Maker
Self wrappers (call C++ stubs)
loaded into Self image
dispatch table entries (select stubs)
linked into virtual machine
C++ stubs (call C/C++ routines)
linked into virtual machine
generated by PrimMaker

Figure 58. Overview of PrimMaker's functionality

When extracting PrimMaker, the main object is `primitiveMaker` and the main method is the one that compiles a template file. PrimMaker constitutes a fair test because it was written before our work on type inference and extraction began. The extractor was nevertheless able to extract PrimMaker unmodified, although we had to make two minor changes elsewhere in the Self world for extraction to succeed (see Section 6.3).

6.1.1 Step 1: grouping objects

The first step, illustrated in Figure 59, is simply the grouping of objects described in Section 4.6. Here we prefer to view it as a separate step, rather than as part of type inference since a later step in extraction (Step 4) reverses the effects of grouping.

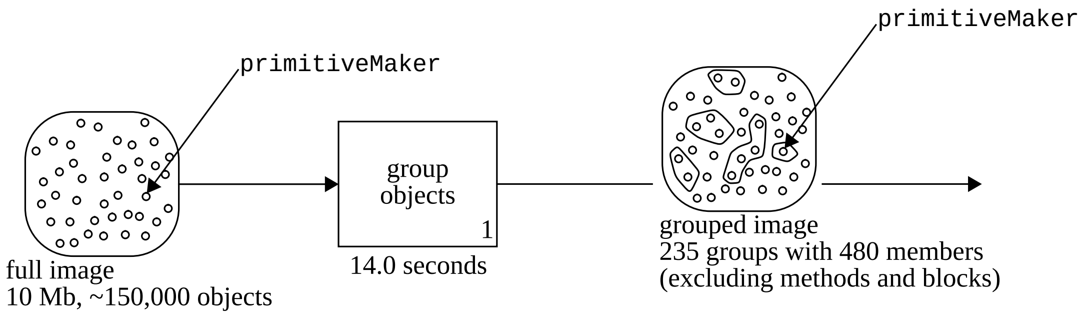


Figure 59. Step 1 of extraction: grouping of objects

Grouping operates "on demand," i.e., processes objects only as the subsequent type inference step encounters them. Hence, most objects in the image are never assigned to a group.

In the specific case of extracting PrimMaker, the grouping step takes 14.0 seconds of CPU time on a 167 MHz UltraSPARC. For the approximately 150,000 objects in the image (counting on an image without the type inferencer), 235 groups containing a total of 480 regular objects are created. In addition, a total of 1,953 groups are created for blocks and 3,745 groups for methods. Thus, the "on demand" grouping avoids processing the majority of objects. Although methods and blocks are just special kinds of objects in Self, the aforementioned numbers for the grouping step specifies methods and blocks separately to give a better picture of how grouping works for regular data objects (under the implemented grouping rules, method and block groups are trivial, always containing exactly one member; see Section 4.6.2).

6.1.2 Step 2: type inference

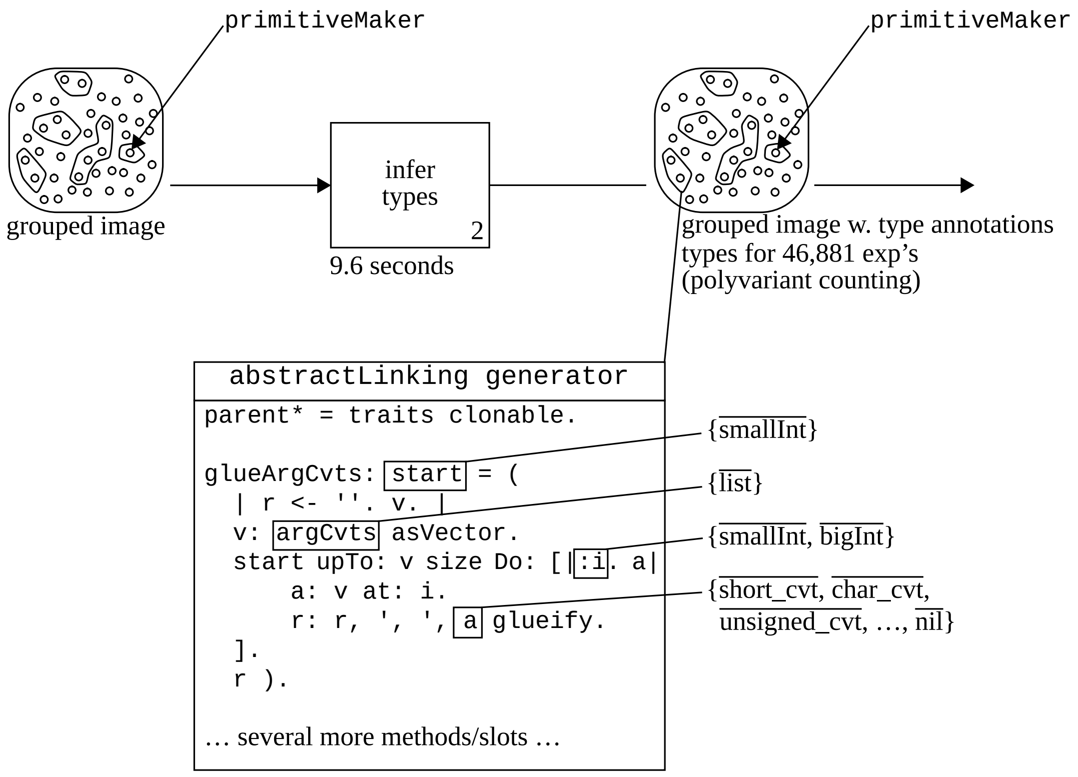
Suppose we need to extract an application that sends the message `size` to the result of some expression `rcvr`. The image in which the application has been developed may contain dozens of methods called `size`, for rectangles, dictionaries, sets, arrays, and other kinds of objects. Which of the many `size` methods should be extracted?

Concrete types provide exactly the information needed to answer this question. For example, if the type reveals that `rcvr` can never return hash tables, there would be no need to extract the size method for hash tables (unless, of course, it was needed elsewhere). Thus, the second step in the extraction of PrimMaker invokes our type-inference algorithm on the main method of PrimMaker. After type inference, every expression and slot in PrimMaker is annotated with a type.

Figure 60 shows the types inferred for the `glueArgCvts:` method found in a certain "generator" object. To avoid clutter, only selected type annotations are shown. Without any further explanation of the role of this method, let us take a look at the annotations computed by the inferencer and see what we can learn. For example, the type of the expression `argCvts` is $\{\overline{\text{list}}\}$. Anticipating the next step, the reader can now see that only the `asVector` method that applies to lists should be extracted (unless, of course, `asVector` is sent to an object with a different object type elsewhere in the application). Likewise, the type of `a` is a set of some 20 object types; thus the `glueify` send is highly polymorphic. This send alone tells us to extract the 20 respective `glueify` methods.

The type inferencer computes sound and conservative types. Soundness enables the extraction algorithm to guarantee full preservation of behavior of the extracted application, as will become clear. Conservatism means that it may sometimes extract objects that will actually not be needed. For example, in Figure 60, the fact that $\overline{\text{nil}}$ is in the receiver type of the `glueify` send can be attributed to conservatism; however, in this case it does not cause extraction of unnecessary methods, since `nil` does not understand `glueify`. We have not yet measured the extent to which conservatism causes extraction of unnecessary objects. It should be possible to do so by applying a coverage tool to the extracted application to determine how much of it is used dynamically over several typical runs.

The type inferencer computes types that are sets of object types. Recall from Section 3.1.2 that object types are groups of initial objects closed under cloning, i.e., if $G \subseteq IM$ is a group, the object type $\overline{G}$ is defined by:


primitiveMaker
primitiveMaker
grouped image
infer types
2
9.6 seconds
grouped image w. type annotations
types for 46,881 exp's
(polyvariant counting)
$\{\overline{\text{smallInt}}\}$
$\{\overline{\text{list}}\}$
$\{\overline{\text{smallInt}}, \overline{\text{bigInt}}\}$
$\{\overline{\text{short\_cvt}}, \overline{\text{char\_cvt}}, \overline{\text{unsigned\_cvt}}, \ldots, \overline{\text{nil}}\}$

```
abstractLinking generator
parent* = traits clonable.

glueArgCvts: start = (
  | r <- ''. v. |
  v: argCvts asVector.
  start upTo: v size Do: [| :i. a|
      a: v at: i.
      r: r, ', ', a glueify.
  ].
  r ).

… several more methods/slots …
```

Figure 60. Step 2 of extraction: inferring types

The type-inference algorithm, when invoked on the main object and method, produces type annotations for all expressions that the application may execute and all slots that it may access.

$$\overline{G} = \bigcup_{\omega \in G} \overline{\omega}$$

where $\overline{\omega}$ denotes the clone family of the initial object ω. On the other hand, since for all initial objects $\overline{\omega} \cap IM = \{\omega\}$, the initial objects in an object type are exactly the members of the corresponding group:

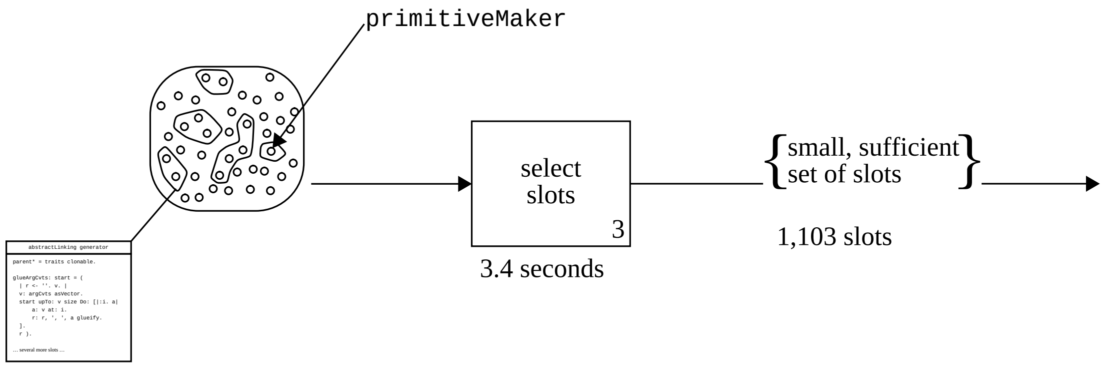
$$\overline{G} \cap IM = G \,.$$

Thus, since the extractor sifts initial objects, it need not distinguish between object types and groups. In other words, the extractor need not be concerned with the objects that the application creates during execution. It needs to extract initial objects only, and the rest will "follow" when the extracted application executes. Consequently, in the following we will primarily use the term "group" rather than "object type" to emphasize that the extractor operates on initial objects.

Type inference of PrimMaker takes 9.6 seconds of CPU time on a 167 MHz UltraSPARC. It computes a type for each of the 46,881 expressions that PrimMaker may execute (counting expressions over templates rather than syntactically).

6.1.3 Step 3: selecting slots

Although a Self image consists of objects, these can be further broken down into slots. An application that uses an object may access only some of its slots. For example, an application may build lists by adding elements one by one, but may never remove elements from these lists. Thus, the application needs the `add:` method but not the `remove:` method for lists. Our extractor uses the type information computed in the previous step to identify a set of slots that is large enough to fully preserve the behavior of the application, yet as small as the available type information permits. We call such a set of slots small and sufficient. Figure 61 illustrates this step.



```
abstractLinking generator

parent* = traits clonable.

glueArgCvts: start = (
  | r <- ''. v. |
  v: argCvts asVector.
  start upTo: v size Do: [|:i. a|
      a: v at: i.
      r: r, ', ', a glueify.
  ].
  r ).

… several more slots …
```

Figure 61. Step 3 of extraction: selecting slots

Based on the inferred types, the extractor selects a set of slots to extract. The set is large enough to preserve the behavior of the application, but as small as the available type information permits.

A small and sufficient set of slots can be computed by simulating each send in the application, determining for each send the slots that might be accessed. Consequently, we attack two subproblems: collecting sends and simulating sends. An efficient implementation can interleave the two subproblems; here we keep them separate for clarity.

Collecting sends. The extractor uses a transitive closure operation to collect a sound (and small) approximation to the set of syntactic sends that the application being extracted may perform. First, it initializes a set with the sends in the main method. Then, for each send in the set, type information leads to the set of methods that may be invoked. The sends in these methods are added to the set and the induction step is repeated until the set of possible sends grows no more. The procedure shown in Figure 62 implements this transitive closure. When invoked on the main method, it returns a list of all sends that the application being extracted may execute.

```
procedure CollectSends(m: method)
   var sends: list;
begin
   “mark m”;  (* To break cycles when processing recursive methods. *)
   for each send in m.code do
      sends.append(send);
      for each receiverGroup in type(send.receiverExp) do
         accessedSlot := receiverGroup.lookup(send.selector);
         if accessedSlot.isMethod and “it is unmarked” then
            sends.append(CollectSends(accessedSlot.contents))
         end
      end
   end;
   return sends
end CollectSends;
```

Figure 62. The `CollectSends` procedure

This recursive procedure collects the set of all sends that an application may execute.

Simulating sends. The extractor now uses the sends to *mark* a small sufficient set of slots. The idea is to simulate the sends one by one. During the simulation of a send, the extractor marks all slots whose presence is needed to preserve the behavior of the send. Specifically, given a send and a possible receiver, it marks the target slots that the send may invoke. In addition, if a target slot is not found directly in the receiver but instead is inherited through a chain of parent links, it marks the parent slots on the inheritance chain leading to the target slot. No further slots are marked.

The above explanation focused (implicitly) on sends that succeed. Consider now failing sends. If a send may fail with message-not-understood error, the simulation of the send discovers this possibility, since it finds no target slot for one or more kinds of receivers. When the simulation finds no target slot, it marks no slots and the send will therefore also fail with message-not-understood in the extracted application. Likewise, if multiple inheritance may cause a send in the unextracted application to fail with ambiguous-selector error because matching slots are inherited from several parents, the simulation detects this situation, and marks all the matching slots and the parents leading to them. Consequently, the simulation forces extraction of enough slots to ensure that the send will still fail with ambiguous-selector error after extraction.

In summary, whether a send succeeds or fails, the extractor marks enough slots to fully preserve the behavior. Since the marked slots preserve the behavior of every send for every possible receiver, and since in Self all computation is performed by passing messages, the behavior of the application as a whole is preserved.

The procedure, `MarkMinSuffSlots`, shown in Figure 63, formalizes the marking of a sufficient set of slots. The set of slots marked is minimal in the restricted sense that no slot can be safely omitted if everything that the type information predicts may happen during execution can in fact happen. In other words: the set of slots marked is as small as the available type information permits. Of course, availability of more accurate type information would allow the extractor to mark a smaller set of slots and still know that it is sufficient (as long as the type information remains sound, a sufficient set of slots is marked).

```
procedure MarkMinSuffSlots(A)
begin
    “clear all slot marks”;
    for each send in CollectSends(A) do
        for each receiverGroup in type(send.receiverExp) do
            “simulate lookup of send.selector starting in receiverGroup”;
            for each matchingSlot “that the lookup found” do
                “set mark in matchingSlot”;
                “set mark in each parent slot on path from receiverGroup
                  to the group containing matchingSlot”
            end
        end
    end
end MarkMinSuffSlots;
```

Figure 63. The `MarkMinSuffSlots` procedure

This procedure marks a sufficient set of slots for an application, given the sends that it may execute.

The only complication that may arise is specific to Self: a lookup may encounter dynamic inheritance. In that case, the extractor cannot apply the standard lookup algorithm. Instead, it uses type information for dynamic parent slots to search *all* possible dynamic parents (analogous to the dynamic dispatch resolver’s handling of dynamic inheritance; see Section 4.1.1).

Given the type information, collecting sends and marking slots takes relatively little time. For PrimMaker, the combined CPU time for executing `CollectSends` and `MarkMinSuffSlots` is 3.4 seconds. 1,103 slots are marked. These slots come from 207 groups that contain a total of 2,639 slots. Thus, basing extraction on individual slots rather than whole objects in this case sharpens the result by avoiding the extraction of 2,639 – 1,103 = 1,536 slots.

In summary, the result of this third step of extraction is a mapping, MarkedSlots, from groups to sets of slots: for a group $G \subseteq IM$, MarkedSlots(G) are the slots in G that were marked by `MarkMinSuffSlots`.

6.1.4 Step 4: selecting objects (ungrouping)

The fourth step, illustrated in Figure 64, determines for each individual object in the image whether or not it needs to be extracted. This is done by carefully "lowering" the group level result from the previous step to the object level.

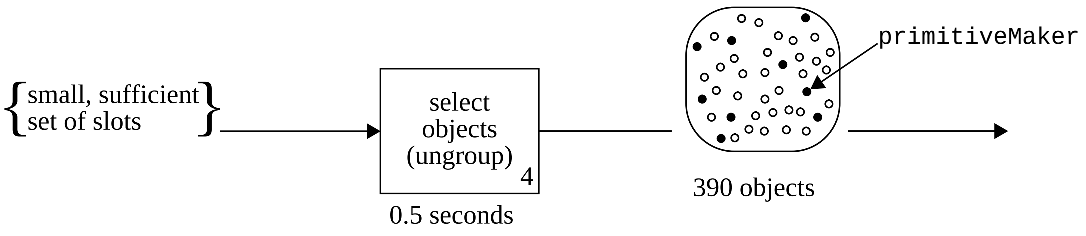


Figure 64. Step 4 of extraction: selecting objects

The extractor maps the group-level slots identified in the previous step back onto the object level.

Let ω be an object and let G = group(ω). If MarkedSlots(G) = Ø, then there is no reason to extract ω, since no slot in ω will ever be accessed by the application. Instead, the extracted version can safely replace any reference to ω by a reference to the empty object. Seen in this light, Step 3 finds those groups from which one or more members must be extracted. A naive way to proceed would then be to extract all the members from the groups which had one or more slots marked, and extract no members from the remaining groups. Succinctly:

$$\text{ExtractSet}' = \{\omega \in IM \mid \text{MarkedSlots}(\text{group}(\omega)) \neq \emptyset\}.$$

This extraction is clearly sound but potentially much larger than need be. For example, just because the application needs *some* point object, it is unlikely that *every* point object in the image will be needed. To extract a smaller but still sufficient set of objects we can view the problem of identifying the required objects as a reachability problem: if the application has no way to reach a specific object, it need not be extracted. This viewpoint leads to a smaller extraction, because the application is limited to accessing objects through the slots that were marked in Step 3.

Definition. Let A be an application. An object is *A-reachable* if and only if:

- it is the main object, or
- it is contained in a slot of an A-reachable object ω, and the slot is in MarkedSlots(group(ω)).

  (It may of course be the case that A will never access a specific slot in the specific object ω, but since we cannot rule out the possibility, we declare the contents A-reachable.)

It is straightforward to convert these rules into executable code, so we omit the step here. Applying the rules identifies a sufficient set of objects:

$$\text{ExtractSet} = \{\omega \in IM \mid \omega \text{ is } A\text{-reachable}\}.$$

Identifying the 390 objects that are PrimMaker-reachable takes only 0.5 seconds, given the MarkedSlots mapping. These objects contain a total of 1,462 marked slots. Comparing this with the 1,103 group level slots in MarkedSlots we conclude that most groups have only a single reachable member. Literal objects, i.e., smallInts, floats, and strings, which are often in large groups, are excluded from ExtractSet, since they do not need to be extracted; instead, they are implicitly created by the virtual machine at start-up time.

For comparison, had the extraction been based on ExtractSet´ (but still counting only the objects grouped in Step 1), the number of objects extracted would be 470 instead of 390. In this specific case, the two numbers differ only moderately because the extraction is based on a non-incremental type inference: few objects besides those involved in PrimMaker have been grouped. In general, viewing the type inferencer as a general purpose resource in the programming environment, many more objects than those being extracted may have be grouped, and the extractor should not rely on this internal property of the type inferencer.

6.1.5 Step 5: dumping the objects

The final step, illustrated in Figure 65, writes out a representation of the objects in the ExtractSet that was identified in the previous step. Writing out a possibly circular structure of objects is a standard problem that has been previously addressed, e.g., by the Smalltalk-80 BOSS system for storing objects in a binary format [91]. In our case, there is only a slight twist to the problem: when writing out an object ω, only the slots given by MarkedSlots(group(ω)) are written out.

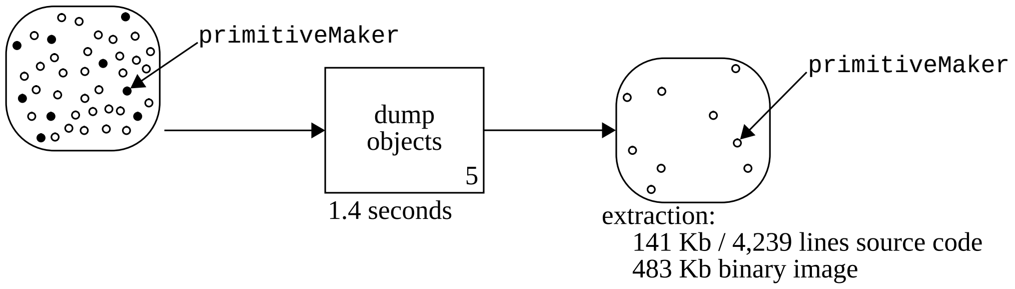


Figure 65. Step 5 of extraction: dumping objects

The final step dumps the slots and objects that were selected by the previous four steps.

The extractor writes a single source file. The file is “stand-alone,” i.e., can be read into an empty virtual machine. Subsequently, a binary image can be easily obtained by simply invoking the primitive that writes out an image.

The extractor’s dump files are somewhat naive compared with programmer-written source code. For example, an extracted list is built slot by slot instead of using the `add:` operation on lists. Consequently, merely looking at the size of the source file can be misleading. Comparing image sizes gives a more consistent measure, although it should also be remarked that Self images tend to be larger than Smalltalk images. since compact image files have never been a priority in the Self project. For PrimMaker, a total of 141 Kb or 4,239 lines of source was dumped in 1.4 seconds. The corresponding image size is 483 Kb, as compared to almost 10 Mb for the original image.

Due to space limitations, we cannot show the entire extracted PrimMaker source file. However, Figure 66 shows the source file produced when extracting HelloWorld from the standard Self image. HelloWorld is simply defined by the following main object:

```
( |
    main = ( 'Hello Self World!' printLine. _Quit ).
| )
```

The main object is easily identifiable in the extracted file (named `obj6`). The boolean objects can also be recognized, although not easily, because they have been pruned quite severely:

- `true` implements `not`, but understands no other messages; not even `ifTrue:False:`.
- `false`, on the other hand, does not understand `not`, but does implement `ifTrue:False:`.

```
_AddSlots: ( | world = (). | )
world _Define: ( |
  obj0 = ().                 "scheduler"
  obj1 = true.               "true"
  obj2 = false.              "false"
  obj3 = ().                 "traits immutableString"
  obj4 = ().                 "system"
  obj5 = self.               "lobby"
  obj6 = ().                 "an object<7>"
  obj7 = byteVector parent.  "traits byteVector"
  obj8 = ().                 "traits orderedClonable"
  obj9 = ().                 "globals"
  obj10 = ().                "traits string"
  obj11 = ().                "traits mutableIndexable"
  obj12 = '' parent.         "traits canonicalString"
  obj13 = ().                "traits collection"
  obj14 = ().                "traits indexable"
  obj15 = ().                "defaultBehavior"
| )
world obj0 _AddSlots: ( |
  stopping <- world obj1.
  isRunning = (  stopping not ).
| )
world obj1 _AddSlots: ( |
  not = world obj2.
| )
world obj2 _AddSlots: ( |
  ifTrue: b1 False: b2 = (  b2 value ).
| )
world obj3 _AddSlots: ( |
  parent* = world obj10.
| )
world obj4 _AddSlots: ( |
  scheduler = world obj0.
| )
world obj5 _AddSlots: ( |
  globals* = world obj9.
  defaultBehavior* = world obj15.
| )
world obj6 _AddSlots: ( |
  main = ( 'Hello Self world!' printLine. _Quit  ).
| )
world obj7 _AddSlots: ( |
  parent* = world obj11.
| )
world obj8 _AddSlots: ( |
  parent* = world obj5.
| )
world obj9 _AddSlots: ( |
  system* = world obj4.
| )
world obj10 _AddSlots: ( |
  asString = (  self ).
  print = (
    scheduler isRunning ifTrue: [stdout write: asString]
                         False: [asString _StringPrint].
    self ).
  parent* = world obj7.
| )
world obj11 _AddSlots: ( |
  parent* = world obj14.
| )
world obj12 _AddSlots: ( |
  parent* = world obj3.
| )
world obj13 _AddSlots: ( |
  parent* = world obj8.
| )
world obj14 _AddSlots: ( |
  parent* = world obj13.
| )
world obj15 _AddSlots: ( |
  printLine = (  print. '\n' print. self ).
| )
world obj 6 main   "Execute extracted application"
```

Figure 66. Source file produced by extractor when extracting HelloWorld

The inheritance path from `traits canonicalString` through `traits immutableString`, `traits string`, `traits byteVector`, `traits mutableIndexable`, `traits indexable`, `traits collection`, and `traits orderedClonable` to the `lobby` can also be traced. This chain is present both because the `printLine` routine is inherited from `defaultBehavior` and because the string printing routine needs access to the `scheduler`, an object in the `globals` name space in the `lobby`. One could imagine collapsing the many vacuous parent chains after extraction. However, larger applications are less likely to have so many of these, so this simplification has not been given priority in our work on extraction.

6.1.6 Summary of the extraction process

The bottom line of the extraction of PrimMaker is a reduction in image size from 10 Mb to 483 Kb, a reduction of 95%. Table 25 summarizes the results of each of the five steps. For each step, the time required and the result produced are listed. For example, the dumping step executes in 1.4 seconds and produces 4,239 lines of stand-alone Self source code (which can be converted to a 483 Kb image). The extraction time is dominated by the grouping and type inference time. However, since grouping and type inference are incremental, it would take less time to redo these steps after a change in the target program. At less than a minute, the total extraction time seems acceptable for a relatively infrequent activity.

	Grouping	Type inference	Selecting slots	Selecting obj's	Dumping obj's
Time[a]	14.0	9.6	3.4	0.5	1.4
Result	480 objects in 235 groups	types for 46,881 exp's	1,103 slots in 207 groups	1,462 slots in 390 objects	141 Kb / 4,239 lines src, 483 Kb image

Table 25. Summary of the extraction of PrimMaker

a. CPU time in seconds on a 167 MHz UltraSPARC.

6.2 Measurements

There are three road blocks that currently prevent us from measuring the performance of the extractor on many applications. First, the Self virtual machine defines some 600 primitives, but the type inference step only supports 200 of these. Second, the Self system contains a scheduler that implements concurrent processes; we have not yet generalized the extractor to cope with concurrency. Third, reflective features of Self and message sends with computed selectors (performs) are only partially supported by the type inferencer and extractor.

Despite these shortcomings, we have been able to extract several test programs. The results are shown in Table 26. The "Type inf. time" column gives CPU seconds for the type inference step. We did not use the incremental feature of the type inferencer: code shared between several applications was analyzed several times. However, we did use incremental grouping; hence, we refrain from giving numbers for the grouping step, since the incrementality makes it difficult to attribute the grouping to specific test programs. The "Extraction time" column reports the CPU time for steps 3, 4, and 5 of the extraction algorithm. "#Objects extracted" reports how many objects where extracted for each application, excluding methods, blocks, and primitive objects (smallInts, floats, and string constants). "Method body size" is the total number of lines in the bodies of all extracted methods. The count includes blank lines and comments (but only when found within method bodies). Block methods are counted as part of the outer method they are nested in to avoid counting them more than once. "Dumped source size" is the total size of the source files that the extractor produced. Finally, "Image size" is the size of the binary image with the extracted application.

Since basic "library code" such as integers, strings, and vectors constitute the majority of the code in many of the smaller extracted applications, they have similar sizes. In particular, all the applications force extraction of the bigInt implementation. BigInts are extracted as a result of analyzing the Self code that handles arithmetic overflow (the code coerces smallInts to bigInts and retries the operation). To quantify the effect of bigInts, and present data that are relevant for comparison against C++ where smallInts do not automatically coerce to bigInts, we repeated all the measurements, this time setting a flag in the type inferencer directing it to consider smallInt overflows a hard error. The results of extracting without bigInts are shown in Table 27. Comparing the two set of measurements, it can be seen that

	CPU seconds on a 167 MHz UltraSPARC					
	Type inf. time	**Extraction time**	**#Objects extracted**	**Method body size**	**Dumped source size**	**Image size**
HelloWorld	0.0	0.0	16	10 lines	2 Kb	136 Kb
Factorial	6.8	2.7	233	1,044 lines	72 Kb	310 Kb
Richards	7.2	3.4	256	1,285 lines	85 Kb	351 Kb
DeltaBlue	10.4	4.2	253	1,359 lines	86 Kb	351 Kb
Diff	15.5	6.5	328	2,064 lines	124 Kb	458 Kb
PrimMaker	9.6	5.3	390	2,242 lines	141 Kb	483 Kb

Table 26. Summary of the extraction of several test programs

bigInts slow down type inference, consistently contribute 450 method body source lines, and cost approximately 50 Kb in image size.

	CPU seconds on a 167 MHz UltraSPARC					
	Type inf. time	**Extraction time**	**#Objects extracted**	**Method body size**	**Dumped source size**	**Image size**
HelloWorld	0.0	0.0	16	10 lines	2 Kb	136 Kb
Factorial	0.9	1.4	230	588 lines	51 Kb	252 Kb
Richards	1.2	1.8	253	829 lines	64 Kb	277 Kb
DeltaBlue	2.3	2.2	250	909 lines	65 Kb	293 Kb
Diff	4.3	3.8	326	1,625 lines	104 Kb	400 Kb
PrimMaker	3.0	3.5	387	1,798 lines	121 Kb	425 Kb

Table 27. Extraction without bigInts

In fairness, it should be mentioned that Self's integer semantics are not just a burden. They do in fact provide significant expressive power and lend conciseness to programs that potentially need to manipulate large integers—conciseness (and robustness) that equivalent C++ programs cannot easily achieve. While it is perhaps unlikely that Diff will be used to compare files with more than 2^{31}[†] lines, one of the extracted programs, Factorial, only works correctly with bigInts (without bigInts it produces an overflow error).

6.3 Discussion

Application extraction raises a number of issues, some specific to Self, some specific to our particular extractor, and others more general. We will first, in Section 6.3.1, discuss issues that impact the design of the extractor. Then, in Section 6.3.2, we delve into the way that the programming style of the application programmer may interact with extraction.

[†] Actually 2^{29} lines, since the Self virtual machine reserves two bits for tagging purposes.

6.3.1 Issues in the design of an extractor

There are three issues that the designer of an extraction algorithm should be aware of. Section 6.3.1.1 discusses the granularity that extraction should be based on, Section 6.3.1.2 discusses how much behavior should be preserved across extraction, and Section 6.3.1.3 considers resources required for extraction.

6.3.1.1 Granularity

Granularity refers to the smallest unit that can be extracted. The larger it is, the more excess baggage may be dragged along when extracting the parts that were deemed necessary to extract. We identify three different granularities that extraction can be based on. From coarser to finer they are:

- *Module-based.* A module-based extractor includes or excludes modules as a whole from the extracted application. A module is a language- (and programming style-) dependent feature, but typically consists of a set of objects and classes. C++, while not employing extraction, uses a module-based approach for including code. For example, if a C++ programmer wants to use a class, he does this by including the module containing the class. The module is typically a ".o-file," and it may contain several other classes and/or objects, all of which will end up in the application.
- *Object/class-based.* An object-based extractor extracts objects (or classes) as a whole. Objects and classes are typically smaller than modules, and so less excess baggage will be extracted along with the necessary objects. For example, ParcPlace Smalltalk implements manual class-based extraction: the programmer can specify a list of classes to be removed from the image [90].
- *Slot/attribute-based.* A slot-based extractor such as our algorithm offers an even finer resolution because it filters out unused methods and variables.

Other granularities are possible, including still finer ones. It may, for instance, be reasonable to eliminate dead code within methods. We have not quantified the difference between the above three granularities with respect to the amount extracted, but we could do so by first computing a set of sufficient slots. Then we can "round off" to whole objects and dump an object-based extraction, and round off to modules and dump a module-based extraction. The object- and module-based extractions obtained in this manner should be very good, since rounding off is done only at the very end, after having delineated using the finer slot-based resolution.

6.3.1.2 How much behavior is preserved across extraction?

Although an extractor should produce the most compact applications it can, an algorithm that extracts the smallest amount of code may or may not be the best choice. While less code extracted is better, the unavoidable consequence may be that less behavior is preserved. One can distinguish between three different levels of behavior preservation. In order of increasing quality they are:

- *Correct programs only.* If the unextracted program executes without error, it is guaranteed that the extracted program will execute without error. No further guarantees apply.

  This level provides the minimally acceptable guarantee, but even so it is dangerous. For example, if there *is* a bug in the application (and isn't there always?), all bets are off when running the extracted application because error conditions may go unnoticed and the application may silently produce erroneous output. In some sense, this minimal degree of behavior preservation is similar to switching off array bounds checking before shipping an application written in, say, Modula-2, a routine practice in our industry. This level may be acceptable when the consequences of unanticipated behavior are small relative to the likelihood of encountering undiscovered bugs.

- *Some error.* If the unextracted program encounters an error, it is guaranteed that the extracted program will also encounter some error, but this may be a different error happening later in the execution.

  Compared with the previous level of behavior preservation, this level has the advantage that extraction will not convert a visible error into a stealthy error. The person running the application will be notified that an error has happened, but no further promises are given. In an extreme case, the application would run for long enough in the erroneous state to output erroneous results.

- *First error*. If the unextracted program encounters an error, the extracted program will encounter exactly the same error.

  This level of behavior preservation facilitates the debugging of errors encountered after delivery (the most expensive kind of errors). It also has the advantage that the extracted application will behave, to the highest possible degree, exactly as the unextracted application, enabling extrapolation of both successful and failing test runs from the unextracted application to the extracted application. For these reasons, we have crafted our extractor to operate at this level.

Our algorithm attains the strongest guarantee because it extracts enough code to preserve the behavior of every send in the extracted application, including the behavior of sends that may fail. We have not yet been able to quantify how much smaller a typical extracted application could be if we were willing to settle with one of the two weaker guarantees. To measure this, we would have had to modify our type-inference algorithm to exploit the increased freedom; this, however, was beyond the initial scope of our work which takes the type-inference algorithm as a given. The trade-offs between different levels of behavior preservation vs. amount of code vs. time to extract merit future exploration.

6.3.1.3 Resources required to extract an application

How good is our extractor? Several criteria could apply. Some are obvious, like speed of the extraction process and memory consumption. All other things being equal, a faster, less memory demanding algorithm is preferable, but it should also be noted that extraction is not like compilation or debugging. The latter activities are performed repeatedly during program development. Extraction, on the other hand, can conceivably be limited to taking place just before delivery, so the resources consumed are less important. Taking a few minutes, our extractor seems to be fast enough to be useful.

Programmer time should also be considered when tallying the cost of extraction. Depending on how powerful the extraction algorithm is, more or less programmer involvement may be required. This issue should not be played down. It has potentially serious consequences, e.g., for reuse: if programmers know that they may have to guide the extractor through code they are about to reuse, they may choose not to reuse, because the effort required to obtain a sufficient understanding of the reused code may exceed the savings from reusing. Although our type-inference algorithm still has room for improvement, it seems to be adequate to drive an extractor that only rarely requires programmer intervention (we give an example of programmer intervention below).

6.3.2 Programming style issues

Should application programmers be worried about extraction while they write programs? Are some styles of programming more conducive to extraction than others? The Self system, having evolved for years without regard to extraction, is a good case to study. Our experience suggests that although extraction can cope with most stylistic variations, two idioms, sends with computed selectors and reflection, can pose problems.

6.3.2.1 Computed selectors (performs)

The `_Perform` primitive sends messages whose names cannot be statically determined by our type inferencer. This lack of information forces the type inferencer to treat it very conservatively; thus potentially forcing the extraction of objects that are not really needed.

The top half of Figure 67 shows a perform from PrimMaker. The performed selector is computed by removing a string from a list and appending a colon to it. The receiver of the perform is `self`, which in this case is `primitiveMaker`, and there is a single argument, `true`. It turns out, but the extractor currently cannot determine this, that there are only three possibilities for the performed selector, so the code shown in the bottom half of Figure 67 is equivalent to that in the top half. Not knowing the performed selector, except that the syntax reveals that it passes one argument, the extractor must assume that *any* 1-argument method in `primitiveMaker` can be invoked. There are 46 such methods, all of which must then be extracted. But then the ball starts rolling and any method or object accessed from any of the 46 methods also needs to be extracted, and so on.

```
self _Perform: (tokenList removeFirst, ':') With: true.
```

```
flag: tokenList removeFirst.   "Get name of flag to set."
flag = 'canFail'        ifTrue: [self canFail: true].
flag = 'canAWS'         ifTrue: [self canAWS: true].
flag = 'passFailHandle' ifTrue: [self passFailHandle: true].
```

Figure 67. A perform (top) and equivalent non-performing code (bottom)

We considered modifying PrimMaker to not use perform, but decided that it was more general to let the extractor accept advice from the programmer. In the specific example, the programmer would specify that the selector performed is one of `canFail:`, `canAWS:`, and `passFailHandle:`. With this advice, precise type inference is again possible. (In other cases, the uncertainty of the performed selector is of minor significance; then the programmer does not have to give any advice.)

If the programmer (inadvertently) were to give unsound advice, the extracted application might not run. Fortunately, it is possible to detect *dynamically* if unsound advice were used during extraction. The idea is to incorporate the advice into the extracted application together with code verifying its soundness. In the above example, the extracted application would check that the performed selector were one of the three that the programmer specified. Should a fourth selector occur, it would of course be too late to repair the damage, but at least it would be possible to output a precise error message.

One alternative to advice-taking would be to improve the type-inference algorithm, possibly by extending it to track values through a constant propagation and/or constant folding mechanism to discover more information about the performed selectors. However, in the present example this approach would not have worked, since, tracing backwards from the perform reveals that the performed selector is read in from a file.

6.3.2.2 Reflection

Normally, the only thing that can be done with an object is to send it a message and observe the result. Sometimes, though, it is necessary to inquire about the structure of an object. For example, the user interface needs to find out the names of an object's slots in order to display it. This manipulation of the structure of an object, known as *structural reflection,* is accomplished in Self via meta-objects called *mirrors.* While working on PrimMaker, the extractor bumped up against two uses of reflection, although neither of them were specific to PrimMaker. They were both found in objects implementing standard data types that are used by most applications.

Reflection first crept into PrimMaker through the `printString` method in collections[†]. PrimMaker prints out list objects during its execution. Lists generate their printString by invoking a general method that applies to several kinds of collections. This general method reflects upon the elements of the collection to see if they implement `printString` and if not, to look them up in a cache of path names for all prototypes. The consequence of this reflection was dire: since our type-inference algorithm does no range analysis, it had to assume that any of the objects in the path cache might be pulled out. This in turn forced extraction of every prototype in the system! It was a one line change to avoid this use of reflection, but it did incur the cost of making the collection `printString` method less robust: a collection with unprintable members is now itself unprintable.

The second way in which PrimMaker uses reflection is related to the previously discussed performs. When a perform send fails, e.g. with message-not-understood, a signal is generated by the virtual machine. The signal takes the form of a message sent to the currently executing process object. When the process object receives the error signal, it calls

[†] The `printString` methods in the Self system have since been modified extensively. Today, printing is (often) based on annotations.

into the reflective domain to handle the signal. While we think it is within reach to extract some reflective code, our type-inference algorithm is currently not able to deal with this specific example.

The aggravating circumstance about both cases described above is that neither was part of the application proper. Rather, they were found in library code that most applications, not just PrimMaker, would end up using. This failure of extraction could severely hinder persons somewhat unfamiliar with Self who tried to extract applications that, say, generated printStrings for sets. How would they feel about getting *every* object in the path cache extracted as part of their applications? Would they understand it? Probably not. Would they find it acceptable? Certainly not. We see no easy solution to this problem, except rewriting existing code on a case by case basis as it is deemed necessary, and encouraging programmers writing new code to keep the limitations of extraction technology in mind.

6.4 Previous and related work

The idea of basing application extraction on type inference goes back at least to 1981 when Borning and Ingalls wrote [18]:

> Also, the type system may help in tracing control flow, thus making it possible to produce application modules containing just the code needed to run a particular application.

We have reviewed Borning and Ingalls' type system in Section 2.3.2.2; unfortunately, they never implemented the envisioned extractor.

ParcPlace Smalltalk, *ObjectWorks Release 4.1*, provides guidance on "deploying an application" [90]. An interactive tool, the Stripper, can be used to remove classes from the system. The Stripper has built-in knowledge of which classes specifically support program development (e.g., the compiler classes), and it can remove these classes from the image. The programmer may specify a list of extra classes to remove. There is no guarantee, however, that an essential class is not removed. Smalltalk/V contains a tool with similar functionality, called the Cloner. Smalltalk/V also has the Object Library Builder. When given a set of root objects and an "import list," the Builder extracts all objects reachable by transitive closure from the roots, up to, but not including, objects on the import list. Typically, the import list consists of standard classes and objects. The Builder does not trace control flow, but computes a simple transitive closure, and hence may extract unused objects [42].

Moore, Wolczko, and Hopkins developed Babel, a Smalltalk to CLOS translator. Babel translates an application found in a Smalltalk image into CLOS source code. Babel includes a "ProgramFinder," a component that decides which methods and classes need translation for a given application. The ProgramFinder performs no global type inference. Instead, it determines the methods that need translation using "implementors"—a Smalltalk facility for finding all classes that define a method with a given selector. Specifically, Babel's ProgramFinder searches for implementors for all message sends in the main method and recurses on all sends in these methods. Moore *et al* point out that the inherent conservatism in implementors results in large translations of programs. They also acknowledge that a type-inference system or type declarations as found in Strongtalk [19] would alleviate this problem. Babel goes beyond a direct translation, aiming to improve performance by replacing message sends with statically-bound function calls when it can determine the invoked method. Babel applies a simple local analysis to conservatively approximate the targets of message sends. Here, again, Babel would obviously be able to benefit from more precise type information to further improve the performance of the translated program. (The simple local analysis yields translated CLOS programs that run two to five times slower than the original Smalltalk programs; however, knowing only execution times, this performance difference can be attributed to a combination of two factors: the translation itself and the efficiency of the Smalltalk and CLOS execution engines.)

Assumpção and Kufuji describe an alternative to extraction [8]. Working in a Self-like language, they *replace* a computer's normal operating system with a Self virtual machine and a Self image. The resulting object-oriented operating system, called Merlin, compares as follows to more conventional operating system like UNIX:

- Merlin favors Self programs since they can execute directly. In contrast, to execute C/C++ programs under Merlin, one needs an "OS library."

- UNIX favors C/C++ programs since they can execute directly (in some sense). In contrast, to execute Self programs under UNIX, one needs a Self virtual machine and an image.

Merlin's forte is that Self applications can share objects straightforwardly since, conceptually, all applications on the machine are extensions to the single Self image residing in the Merlin kernel. Consequently, the need for extraction to conserve space diminishes, since the size of the image in the kernel can be amortized over all applications in the Merlin system. Still, for embedded systems, a Merlin-wide extraction may be beneficial to reduce the size of the system as a whole. The main drawback of Merlin is its need to take over the entire computer, similar to the philosophy behind Smalltalk-72 [40]. While early Smalltalk programmers accepted this sacrifice, today most users may be less willing to give up their normal computing environment to run Self programs. Hence, an interesting direction for future work may be to build support for Merlin-style sharing of objects in conventional operating systems.

The Lucid Common Lisp system included Treeshake, a delivery toolkit module that attempted to extract applications by decompiling them and discovering the interconnections between modules [16]. It made no attempts to resolve generic functions according to the types that would be used at run time; all arms of any generic function used would be included. In the end, it was not considered an unqualified success [126].

Allegro Common Lisp used a different strategy for application delivery: instead of extracting an application, it could start an application running in an empty world and lazily load in modules as needed [16, 126]. Although this technique can reduce the memory footprint, it would seem to still require as much disk space as the full system. Furthermore, it may increase start-up time of the application.

Palsberg and Schwartzbach describe an algorithm for eliminating dead code [88]. Like our algorithm, the core of their algorithm is a type-inference system that enables conservative control-flow analysis. The environments are different, however. Our algorithm analyzes an image, sifting through objects (and code) to determine what can be safely omitted. Their algorithm analyzes a textual representation of a program to find code that will never be executed. Another difference is that their algorithm works on programs written in BOPL (Basic Object Programming Language), a minimal object-based language designed for teaching and studying programming language issues. While BOPL contains the essentials of real object-oriented languages, the minimality of the language and implementation makes it unsuitable for writing large programs. Thus, Palsberg and Schwartzbach did not have the luxury of a large body of existing code to test their algorithm on.

Statically-typed object-oriented languages such as Beta, C++, and Eiffel typically rely on "traditional" delivery tools inherited from procedural languages. Usually, modules are compiled into ".o-files," possibly combining these into static or dynamic library files. Applications are then built by invoking a linker such as `ld` under UNIX. The linker starts with a specified set of .o-files and computes the transitive closure of all references in these files and any referenced .o-files in libraries. Then, all reachable .o-files are concatenated to form the application.

This traditional approach differs from the Self/Smalltalk situation in that there is no initial confusion between the programming environment and the application. While it is not necessary to carve the application out of a massive development environment (it is just built from the parts that the programmer or compiler specifies), unreachable code may still enter the final application—and increasingly so, as programs and libraries get larger and code reuse becomes more common. According to a recent study by Srivastava who analyzed several large C++ programs, such programs contain significantly more unreachable code than programs written in procedural languages like C and Fortran. Using a simple static analysis algorithm—one which constructs a call graph under the assumption that all virtual methods of all instantiated classes may be invoked—he found that up to 26% of the code in large C++ programs is unreachable [109]. We believe that by applying a precise type-inference algorithm, an even larger amount of code can effectively be proven dead, and thus safely omitted from the linked applications, a point that Sakkinen [102] also makes in his reply to Srivastava's paper. Moreover, as the current trend towards use of frameworks in the development of applications continues, the amount of dead code will likely increase further.

6.5 Summary

The problem of delivering modestly-sized applications without the overhead of the development environment has hindered the acceptance of integrated object-oriented programming environments like Smalltalk-80, Self, and CLOS.

Working in Self, we have demonstrated that with the advances in type inference, it is now possible to construct an automated application extractor.

Although more experience is needed, the early results are promising: several medium-sized programs have been extracted in a few minutes of CPU time. The resulting images are often an order of magnitude smaller than the full development environment.

To be useful, extraction must preserve the behavior of the extracted application. We identified three different levels of behavior preservation: correct programs only, some error, and first error. Our extractor provides the strongest guarantees: even in the presence of bugs in the application, extraction will not alter the behavior.

We identified three granularities that extraction can be based on: modules, objects, and slots. Our extractor operates at the finest granularity, that of slots, in order to winnow out as much unneeded code as possible.

Our extractor operates without human intervention most of the time. However, two programming idioms can confound it: sends with computed selectors (performs) and reflection. Currently, programmer intervention may be called upon in such cases.

7 Other applications of type information

The preceding chapter described how type inference can support application extraction. This chapter presents three additional applications of type information in object-oriented programming environments: static checking (Section 7.1), browsing (Section 7.2), and optimizing compilation (Section 7.3). Proof-of-concept versions of tools performing these tasks have been implemented in cooperation with other researchers.

7.1 Static checking

The difficulty of eliminating message-not-understood errors in shipped applications has often been mentioned as an argument against using dynamically-typed object-oriented languages for production programming[†]. With type inference, this concern can be addressed. Using the inferred types, a *checker* can statically determine where in applications messages may not be understood or where certain other errors may happen during execution. The programmer can then inspect these places, deciding for each case whether the raised alert

- reflects a flaw in the program, or
- was caused by the inevitable conservatism of the checker.

A reasonable action in the latter situation might be to add an annotation to the program to document the case for future readers and to suppress the alert in subsequent invocations of the checker. The programmer, of course, may also choose to ignore the alert completely. In this regard, the use of the type-inference-based checker resembles the use of `lint`, an optional static checker for C programs [64].

Together with Ole Lehrmann Madsen, we have implemented an experimental version of such a type-inference-based checker for Self programs. Section 7.1.1 describes the user interface of the checker. Subsequently, Section 7.1.2 describes how the checker detects potential message-not-understood errors, and Section 7.1.3 briefly outlines how other kinds of errors may be detected statically. Finally, Section 7.1.4 discusses related work.

7.1.1 The checker's interface

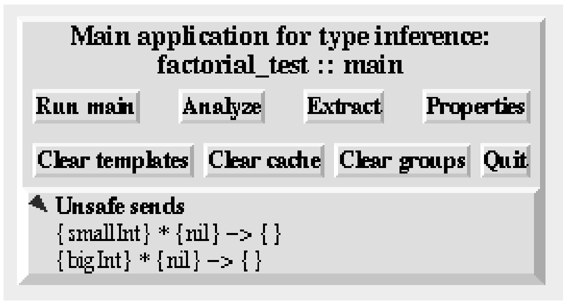
Figure 68 shows the checking tool's appearance in the Self 4.0 programming environment. The top part of the object defines the interface to the type inferencer as a whole. It displays the application currently attached to the type inferencer, in this case the `main` method of some `factorial_test` object; this method simply computes the factorial of a smallInt. The top part of the type inferencer object also provides buttons for executing, analyzing, and extracting the application and several buttons for controlling properties of the type inferencer itself. The bottom part, "Unsafe sends," constitutes the check tool per se. It displays the results of invoking the static checker on the current application: a list of all the templates containing "unsafe sends" (for implementation reasons, object types in the figure are shown without the bar). In Section 7.1.2, we discuss several possibilities for what may constitute an unsafe send. For now, it suffices to think of it as a send that may result in a message-not-understood error.

The templates in the list of unsafe sends are hyperlinks: activating a link opens a type-based browser (described in Section 7.2) on the method in question. The browser presents the method where the potential error surfaced. The cause of the error may be found in a different method which the programmer can locate by using the capabilities of the browser.

In the application shown in Figure 68, the checker found two templates that contain unsafe sends. The templates, presented as

$\{\overline{\text{smallInt}}\}$ `*` $\{\overline{\text{nil}}\} \rightarrow \{\}$
$\{\overline{\text{bigInt}}\}$ `*` $\{\overline{\text{nil}}\} \rightarrow \{\}$

[†] However, it is commonly countered, this possibility is no worse and no more likely than the possibility of lurking logical errors in the application, be it written in a statically-typed or dynamically-typed language.


Main application for type inference:
factorial_test :: main

Run main
Analyze
Extract
Properties
Clear templates
Clear cache
Clear groups
Quit

Unsafe sends
```
{smallInt} * {nil} -> { }
{bigInt} * {nil} -> { }
```

Interface to type inference system as a whole.
Result of applying the static checker (list of templates containing unsafe sends).

Figure 68. The type inferencer's user interface and the result of invoking the static checker

represent attempts to multiply a smallInt and a bigInt with `nil`. The empty result types on the right hand side of the arrows furthermore reveal that these attempts cannot succeed, since there are no possible results. Why are these templates unsafe? After all, both smallInts and bigInts understand the multiplication message "`*`". The templates are unsafe because they *contain* sends that may fail, as will become clear in Section 7.2.

7.1.2 Checking for message-not-understood errors

The current implementation of the static checker is experimental and does not realize the full potential for static checking that the available type information permits. It checks for message-not-understood errors only. While these errors are the most important to check for in Self programs, there are other possibilities which will be discussed below.

To detect potential message-not-understood errors, the checker processes the send expressions one at a time, using the same transitive closure procedure as the extractor to collect all the send expressions that the application may execute; see Section 6.1.3. For each send expression, the checker determines whether the send never fails, may fail, or always fails. To facilitate the following explanation, we express these three possibilities in terms of colors; see Table 28. (The browser, described in Section 7.2, also uses these colors.)

Color	**Classification**	**At run time:**
green	the send never fails	"proceed at full speed"
amber	the send may fail	"proceed with caution"
red	the send always fails	"stop!"

Table 28. The three outcomes of statically checking a send expression

Suppose the checker is processing a send expression with selector *S* and receiver type $\{\rho_1, \rho_2, ..., \rho_n\}$. To determine the send expression's color, the checker simulates a lookup of the selector *S*, starting in each of the possible receiver object types, ρ_1, ρ_2, ..., ρ_n. For simplicity, assume for a moment that dynamic inheritance does not occur; then lookups can be performed in the domain of object types with the same results as if they were performed in the domain of objects. When processing a receiver object type, ρ_i, the checker determines which color it *calls for*:

- If ρ_i understands the message *S*, it calls for green.
- If ρ_i does not understand the message *S*, it calls for red.

Having determined the color that each receiver object type calls for, the checker computes the color of the send expression:

- If all of $\{\rho_1, \rho_2, ..., \rho_n\}$ call for green, the send is green (the send never fails since all the possible receivers understand the message).

- If all of $\{\rho_1, \rho_2, ..., \rho_n\}$ call for red, the send is red (if executed, the send always fails since none of the possible receivers understand the message).

- If some of $\{\rho_1, \rho_2, ..., \rho_n\}$ call for red and some call for green, the send is amber (according to the available information, the send can both fail and succeed).

Dynamic inheritance complicates the checking of send expressions slightly. When a lookup through one of the receiver object types, say ρ_i, encounters an assignable parent slot, the checker uses the type inferred for this slot to conservatively reason about all possible cases that may occur at run time. This approach is similar to the type-inference-time lookup algorithm described in Section 4.1.1. Specifically, when facing an assignable parent slot for some receiver object type ρ_i, the checker distinguishes three cases:

- If the lookup succeeds for all of the possible dynamic parents (according to the type of the assignable parent slot), ρ_i calls for green.

- If the lookup succeeds for none of the possible dynamic parents, ρ_i calls for red.

- If the lookup succeeds for some of the possible dynamic parents and fails for others, ρ_i *demands* the color amber for the send expression—regardless of which colors the other receiver object types call for.

We have explained how the checker collects sends and how it checks each one. It is then a simple matter to collect and display a list of the templates containing unsafe sends. The currently implemented checker displays templates containing red sends only. A more complete implementation would allow the programmer to select between viewing templates containing sends that are red, amber, or of either color.

Table 29 shows measurements obtained by checking several benchmarks. As the numbers indicate, with the available precision of type inference, there are no red alerts. However, amber sends occur with increasing frequency as the benchmarks get larger (the benchmarks in the table are ordered according to size in terms of lines of method source; see Table 7 on page 65). It is safe to assume that these benchmark programs have been reasonably well debugged, so where do these amber alerts come from? Although we have still to perform an exhaustive inspection, we speculate that there are several reasons for the amber sends:

- Even though the benchmarks execute without message-not-understood errors, it is possible that some of the amber sends reflect flaws that should be corrected.

- The majority of the amber sends are probably caused by the type inferencer's lack of precision when analyzing code with data polymorphism. The increasing frequency of amber sends as the size of the benchmark increases supports this explanation, since a larger application is more likely to use a given kind of object in more than one manner. For example, it is more likely that a large application builds several kinds of lists (lists of booleans, lists of floats, lists of integers, etc.) than it is for a small application, which typically use lists for a single task only.

- A smaller, but still significant number of amber sends may result from imprecisions caused by insufficiently accurate flow analysis of variable accesses. Recall from Section 4.5 that the implemented type inferencer does not use full def–use chaining; it performs a less powerful SSA transformation and only on local variables. Section 7.2 below presents in detail an example from `factorial_test` where lacking precision of flow-analysis of variable accesses results in an amber send.

- A small fraction of the amber sends, including a few in PrimMaker, result from the type inferencer's treatment of the `_Perform` primitive. Since the type inferencer has no information about the performed selector, it conservatively assumes that the perform may fail, i.e., it is colored amber.

We suspect that most of the amber sends are caused by imprecisions in the type inferencer relating to data polymorphism and flow-sensitive analysis of variable accesses. Such imprecision would never create a red send, but it may turn a green send into an amber one by polluting the inferred type of the receiver with additional object types. Thus, implementing the techniques described in Chapter 4 to strengthen these aspects of the type inferencer would reduce the number of false alerts raised by the checker. Even so, some sends that do not actually produce message-not-understood errors when the program is executed will likely remain amber, or perhaps even red. In this regard, checked Self programs resemble Beta programs, which are mostly statically typed but in some places require dynamic type

	# red	# amber	# green
HelloWorld	0	0	18
Factorial	0	4	3,262
Richards	0	41	4,005
DeltaBlue	0	51	4,142
Diff	0	102	6,162
PrimMaker	0	227	6,421

Table 29. Results of statically checking for message-not-understood

checking [76]. However, dynamic typing may occur more often in Self programs, since at least the existing ones have been written with little attention to the possibility of statically checking for message-not-understood errors.

7.1.3 Checking for other kinds of errors

In addition to checking for message-not-understood, the information computed by the type inferencer allows static checking for other kinds of errors, some of which fall outside the range of checks supported by typical static type systems. The following is a partial list.

Multiple inheritance ambiguities. Self and other dynamic languages supporting multiple inheritance permit the construction of inheritance graphs that contain ambiguities. An ambiguity has no effect until an object receives one of the messages for which there is an ambiguity. Then a run-time error results.

Although one could statically check for, and disallow, the *construction* of inheritance graphs containing ambiguities for any selector, this approach was not chosen for Self: it would (unnecessarily) restrict the expressive power of the language, and it would conflict with dynamic inheritance (performing the check upon each assignment of a parent pointer incurs considerable cost on switching parents).

Since the type information provided by the inferencer conservatively approximates which messages are sent to which kinds of objects, type inference enables a less obtrusive static test for ambiguities. For each kind of object, this ambiguity test can target the selectors predicted by type inference, while not faltering the application on “innocent” ambiguities, i.e., those that cannot affect the execution because the object containing them never receives the given message. Moreover, using type information for assignable parent slots, the ambiguity checker can reason conservatively about dynamic inheritance, statically catching ambiguities caused by switching parent pointers. We omit the details of detecting potential ambiguities based on type information, but note that the problem is similar to detecting message-not-understood errors: the ambiguity checker also operates by simulating lookups.

Non-LIFO blocks. Self does not permit the invocation of block closures after their lexically enclosing scope has terminated. Any attempt to do so results in a run-time error. Type inference can support static detection of potential non-LIFO invocations:

- First, type inference helps recognize closures† at risk: unless a closure may be stored in an instance variable or returned by some method, it cannot outlive its lexically enclosing method, and thus faces no danger of non-LIFO invocation.
- Second, using the call graph built by the type inferencer, a conservative test can be applied to invocations of risky closures to determine if any of them might constitute a non-LIFO invocation: an invocation may be non-LIFO if the call graph contains a path from the main template to the invoking send not passing through the template that represents the lexical scope of the invoked closure. (This path corresponds to a call stack configuration with the

† Actually, type inference helps recognize *closure object types*, not closures (see Section 4.4). However, for conciseness and simplicity, we use run-time terminology here.

closure invocation at the top and no activation record for the closure's lexical scope on the stack; hence, the invocation must be non-LIFO.)

The second test generalizes to deeply nested blocks at the slight complication of looking for paths from the main template to the invoking send that bypass one or more of the templates on the invoked closure's lexical chain.

The current implementation of Self disallows non-LIFO blocks because these might slow down calls [58] (Section 6.3.3). Since type inference enables static recognition of non-LIFO closure invocations, it may also play a role in adding support for non-LIFO blocks to Self. As long as most cases can safely be declared LIFO, additional resources can be spent on the remaining few cases that may need the non-LIFO support. How well such a static implementation technique would compare against the dynamic techniques of Deutsch and Schiffman [41] remains an open question.

Detecting access to uninitialized variables. In some languages, including C++ but not Self, variables contain undefined initial values. To ensure robustness of programs, access to uninitialized variables must be detected and prevented. The flow information provided by type inference can support conservative detection of such accesses. In Self, however, this problem does not exist. Since the initial value of a slot defaults to `nil`, which understands very few messages, accesses to uninitialized slots usually show up when checking for message-not-understood errors.

7.1.4 Related work

The idea of using static analysis to detect potential errors in programs did not originate with type inference. Even compilers for statically-typed languages use a simple local analysis when checking types, since most expressions in such languages have no declared type. For example in Pascal programs, variables, parameters, and results of procedures have a declared type, but the types of expressions are inferred using a simple local inference system. However, since the inference system is so simple, a Pascal type checker has little in common with our static checker for Self. More similar in spirit to our static checker—both are based on extensive global analysis—is the pioneering work by Fosdick and Osterweil [43]. They applied interprocedural data-flow analysis to detect anomalies such as references to uninitialized variables and useless definitions of variables in Fortran programs.

Most statically-typed languages, including C++ and Eiffel, use a binary model for type checking: a statement either type-checks or it does not (in which case the program cannot be executed). The Beta compiler, however, resembles our checker in distinguishing between three different cases [76]: type safe, need run-time type check (so-called qua check), and type error.

As explained in Section 2.3, some type-inference systems, including Typed Smalltalk and ML's inferencer, combine type inference with proving the absence of run-time type errors: type inference succeeds only for type-correct methods. This approach has the merit of eliminating the need for a separate checking tool. However, the separation of concerns that we have opted for has advantages too: type inference to a higher degree becomes a multipurpose resource instead of favoring one particular application, and an existing body of code can be handled more adequately when inference does not imply full static type checking.

Overall, static checking based on type inference has the potential to allow precise and unobtrusive early detection of many errors in applications written in dynamically-typed programming languages. It can also detect certain kinds of errors that type systems in statically-typed languages commonly cannot target, such as access to uninitialized variables in C++ programs.

7.2 Browsing

Recent publications have argued that object-oriented programs in some regards are harder to understand, debug, and maintain than procedural programs. For example, Wilde, Matthews, and Huitt state [128]:

> For the maintainer, dynamic binding and polymorphism are two-edged swords. They give programs the flexibility that is a main objective of object-oriented programming, but they also make programs harder to understand.

Ponder and Bush raise the same point [97]:

Like *goto*'s, polymorphism can be useful if applied properly. But it creates a Catch-22 problem for anyone trying to read a program: the operations performed are determined from the variable type, and the variable types are deduced from the operations.

To perform a maintenance task, the programmer must understand the code pertaining to the task. Usually, this code is dispersed over several methods, requiring the programmer to trace control flow through calls from method to method. Studies have shown that this tracing can be error prone and time consuming [127]. In object-oriented programs, the tracing may be harder than in procedural programs. First, since object-oriented design practices encourage factoring code into smaller methods to increase reuse and encapsulation, code for a given task becomes dispersed over more methods, forcing the programmer to trace through more calls to cover the relevant code. Second, dynamic dispatch makes it harder to trace through each call, since the target(s) are non-manifest. For the latter problem, static type declarations help some since they limit the possible targets for dynamically-dispatched sends. In general, however, static declarations will not (and should not) narrow down the possible choices to a single method; polymorphism and reusability of code would then suffer too much.

Wilde and Huitt [127] envision four approaches to help the programmer manage dynamic dispatch in object-oriented environments:

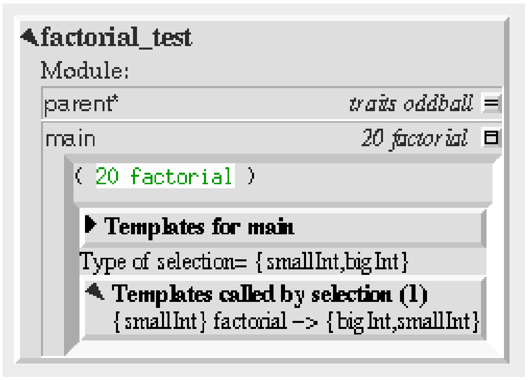
- *Worst-case*: conservatively resolve dynamic dispatch to all methods with the same selector as the send (like "implementors" in Smalltalk). For unusual selectors, this approach works well, but for common selectors, the programmer may be overwhelmed by many irrelevant potential targets.
- *Dynamic analysis*: monitor the classes of receivers that occur during execution (akin to type feedback; see Section 2.3.2.12). However, dynamic analysis is inherently unsound: if a particular case has not occurred during test runs, it will be missed.
- *Human input*: allow the user to describe the expected classes of objects. The potential for human error makes this approach unsound.
- *Static analysis*: this technique can potentially resolve dynamic dispatch more precisely than the worst-case approach. They mention Typed Smalltalk (see Section 2.3.2.4) as promising, but give no concrete results.

The type-inference system we have built is well-suited for supporting the fourth approach: the concrete types computed by the inferencer enable a browser to soundly and precisely resolve dynamic dispatch. Type inference, however, can assist the programmer in more ways than by resolving dispatch. Palsberg, Schwartzbach and the present author describe in [6] how a type-based "hyper-browser" can:

- *Trace forward control flow*, i.e., given a send expression locate the methods it may invoke.
- *Trace backward control flow*, i.e., given a method locate the send expressions that may invoke it (and possibly present only those supplying arguments of certain types).
- *Trace data flow*, e.g., determine *how* a given method can return a certain kind of object (say, `nil`), or find all assignments to a given variable.
- *Show types* for expressions and variables.

Together with Ole Lehrmann Madsen, we have built a proof-of-concept version of such a type-based browser for Self programs. The browser, an extension of Madsen's syntax-directed editor [74], provides the usual features of a browser such as viewing and editing of source code in addition to the type-based capabilities. The browser has been integrated with the static checker described in Section 7.1. It displays source code in the colors computed by the checker, and the list of unsafe sends found by the checker can be used as a starting point for browsing a program.

Figure 69 shows a snapshot of a method opened with the browser. The method is the main method of the `factorial_test` application that was checked in Figure 68 on page 150. The browser resides within an outliner, the standard way that objects are displayed in the Self 4.0 programming environment [119]. The main method contains only a single send expression: `20 factorial`. On a color screen, this expression appears in green, indicating that it cannot cause a message-not-understood error. Below the method body, the browser displays information in part derived from the types computed by the inferencer:



```
factorial_test
Module:
parent*                          traits oddball
main                             20 factorial
  ( 20 factorial )
  Templates for main
  Type of selection= {smallInt,bigInt}
  Templates called by selection (1)
    {smallInt} factorial -> {bigInt,smallInt}
```

Figure 69. `factorial_test`'s main method shown in the type-based browser

The expression `20 factorial` has been selected (highlighted background). Below the body of the method, the browser shows the type of the selection, {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}, and which method(s) it may invoke, in this case the `factorial` method that applies to smallInts.

- *Templates for main.* This panel, while not expanded in the figure, accumulates a stack as the programmer navigates through the program, allowing him to return to previously visited methods.
- *Type of selection.* As indicated by the highlighted background, the programmer has selected the `20 factorial` expression in the main method. In response, the browser displays the type of the expression, in this case {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}.
- *Templates called by selection.* The bottom panel shows the methods that the currently selected send expression may invoke. In the present case, the selected send can invoke only the `factorial` method for smallInts.

The list of "templates called by selection" are hyperlinks. Activating a link opens a browser on the method for the given template. For example, if the programmer activates the link below the main method of `factorial_test`, a browser is opened on the `factorial` method, as shown in Figure 70. On a color screen, all sends in the `facto-rial` method are green. In addition to the navigation stack below the code of `factorial`, there is now a new panel:

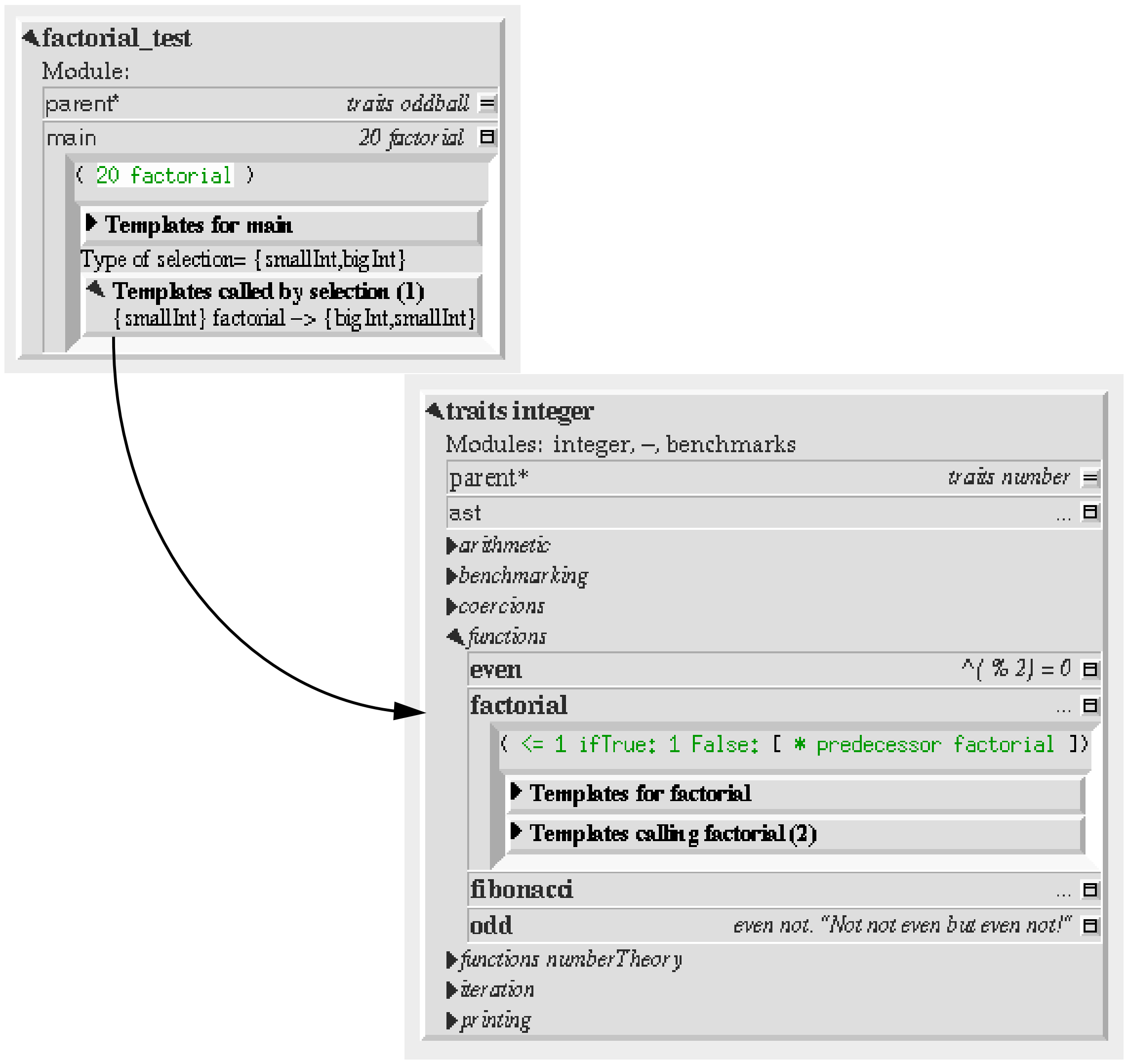
- *Templates calling* `factorial`. This list, collapsed in the figure, allows navigation in the reverse direction of control flow: it contains all templates in the application that may call the `factorial` method.

Our final example demonstrates how the type-based browser may be used to locate at potential error. The paragraph numbers below refer to the numbers annotating parts of Figure 71.

1. Assume the programmer wants to find the reason for the existence of the unsafe template {$\overline{\text{bigInt}}$} `*` {$\overline{\text{nil}}$} $\rightarrow$ {}. The programmer selects it in the checker's interface by clicking on the line next to the "1" annotation in Figure 71.

2. In response, the system opens a type-based browser on the "`*`" method in `traits bigInt`. Watching the code of this method, one send immediately stands out: `a multiplyBigInteger:` is red (and the only red send), because the object `a`, which is `nil`, does not understand `multiplyBigInteger:`. While the symptom, i.e., the red send, is found in the "`*`" method, the problem is not: `a` is an argument of the "`*`" method. At this point the programmer could use "senders" to find a list of all places in the image that sends "`*`" and look through these to determine which of them might supply `nil` as the argument. However, this approach is inefficient since the message "`*`" is common: some 300 methods send "`*`" in Self 4.0.

3. The type-browser, however, can be more selective than the text-based senders tool. Opening the "Templates calling `*`" panel reveals a list of all places in the current application that may invoke the "`*`" method in `traits bigInt`



```
factorial_test
  Module:
  parent*                                   traits oddball
  main                                      20 factorial
    ( 20 factorial )
    Templates for main
    Type of selection= {smallInt,bigInt}
    Templates called by selection (1)
      {smallInt} factorial -> {bigInt,smallInt}
```

```
traits integer
  Modules: integer, –, benchmarks
  parent*                                   traits number
  ast                                       ...
  arithmetic
  benchmarking
  coercions
  functions
    even                                    ^( % 2) = 0
    factorial                               ...
      ( <= 1 ifTrue: 1 False: [ * predecessor factorial ])
      Templates for factorial
      Templates calling factorial (2)
    fibonacci                               ...
    odd                  even not. "Not not even but even not!"
  functions numberTheory
  iteration
  printing
```

Figure 70. Navigating from a send to the invoked method

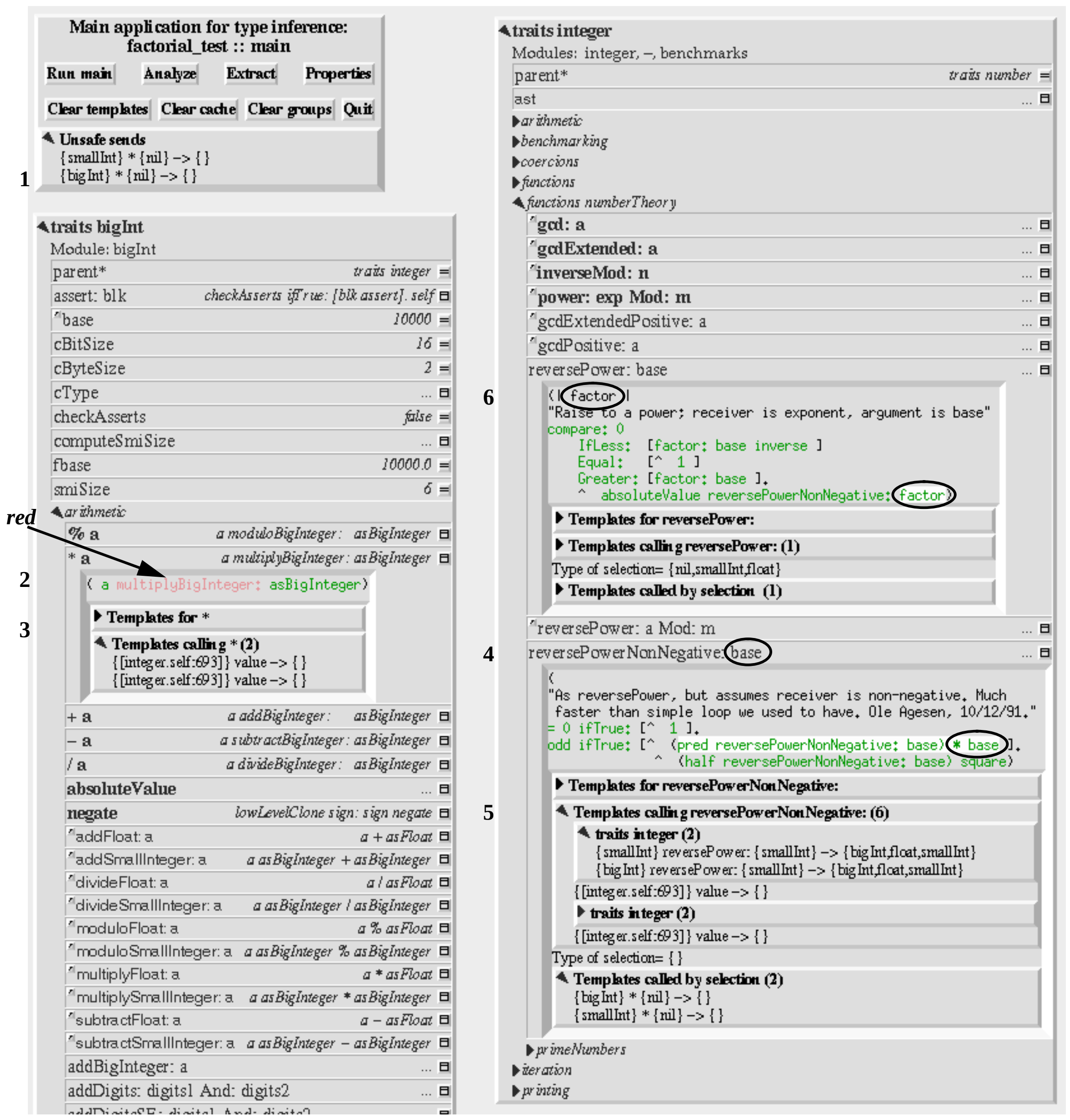
on a bigInt receiver with `nil` as the argument. Instead of hundreds of possibilities, this list contains only two templates, both of which are derived from a block in the integer module.

4. Next, the programmer opens a browser on the method containing the block found in Step 3. This method, `reversePowerNonNegative:`, contains a single multiplication,

```
    (pred reversePowerNonNegative: base) * base
```

whose argument `base` is itself an argument of `reversePowerNonNegative:`. Thus, the programmer must trace control flow backwards once more, in the hunt for the error. However, before leaving the `reversePowerNonNegative:` method, we can note that the selected multiplication expression indeed is responsible for generating the {$\overline{\text{bigInt}}$} `*` {$\overline{\text{nil}}$} → {} template. The "Templates called by selection (2)" field in the `reversePowerNonNegative:` part of the Figure 71, shows that the selected expression (i.e., the above multiplication) may invoke the {$\overline{\text{bigInt}}$} `*` {$\overline{\text{nil}}$} → {} template.

5. Going through the list of "templates calling `reversePowerNonNegative:`" the programmer can immediately eliminate the two block method templates; these are the recursive calls found in `reversePowerNonNegative:`



```
Main application for type inference:
factorial_test :: main
Run main   Analyze   Extract   Properties
Clear templates   Clear cache   Clear groups   Quit
Unsafe sends
  {smallInt} * {nil} -> { }
  {bigInt} * {nil} -> { }

traits bigInt
Module: bigInt
parent*                                  traits integer
assert: blk            checkAsserts ifTrue: [blk assert]. self
base                                              10000
cBitSize                                             16
cByteSize                                             2
cType                                               ...
checkAsserts                                      false
computeSmiSize                                      ...
fbase                                           10000.0
smiSize                                               6
arithmetic
  % a              a moduloBigInteger:   asBigInteger
  * a              a multiplyBigInteger: asBigInteger
    ( a multiplyBigInteger: asBigInteger)
    Templates for *
    Templates calling * (2)
      {[integer.self:693]} value -> { }
      {[integer.self:693]} value -> { }
  + a              a addBigInteger:      asBigInteger
  - a              a subtractBigInteger: asBigInteger
  / a              a divideBigInteger:   asBigInteger
  absoluteValue                                     ...
  negate               lowLevelClone sign: sign negate
  addFloat: a                          a + asFloat
  addSmallInteger: a      a asBigInteger + asBigInteger
  divideFloat: a                       a / asFloat
  divideSmallInteger: a   a asBigInteger / asBigInteger
  moduloFloat: a                       a % asFloat
  moduloSmallInteger: a   a asBigInteger % asBigInteger
  multiplyFloat: a                     a * asFloat
  multiplySmallInteger: a a asBigInteger * asBigInteger
  subtractFloat: a                     a - asFloat
  subtractSmallInteger: a a asBigInteger - asBigInteger
  addBigInteger: a                                  ...
  addDigits: digits1 And: digits2                   ...

traits integer
Modules: integer, –, benchmarks
parent*                                   traits number
ast                                                 ...
arithmetic
benchmarking
coercions
functions
functions numberTheory
  gcd: a                                            ...
  gcdExtended: a                                    ...
  inverseMod: n                                     ...
  power: exp Mod: m                                 ...
  gcdExtendedPositive: a                            ...
  gcdPositive: a                                    ...
  reversePower: base                                ...
    (| factor |
    "Raise to a power; receiver is exponent, argument is base"
    compare: 0
        IfLess:  [factor: base inverse ]
        Equal:   [^  1 ]
        Greater: [factor: base ].
        ^  absoluteValue reversePowerNonNegative: factor)
    Templates for reversePower:
    Templates calling reversePower: (1)
    Type of selection= {nil,smallInt,float}
    Templates called by selection  (1)
  reversePower: a Mod: m                            ...
  reversePowerNonNegative: base                     ...
    (
    "As reversePower, but assumes receiver is non-negative. Much
     faster than simple loop we used to have. Ole Agesen, 10/12/91."
    = 0 ifTrue: [^  1 ].
    odd ifTrue: [^  (pred reversePowerNonNegative: base) * base ].
                 ^  (half reversePowerNonNegative: base) square)
    Templates for reversePowerNonNegative:
    Templates calling reversePowerNonNegative: (6)
      traits integer (2)
        {smallInt} reversePower: {smallInt} -> {bigInt,float,smallInt}
        {bigInt} reversePower: {smallInt} -> {bigInt,float,smallInt}
      {[integer.self:693]} value -> { }
      traits integer (2)
      {[integer.self:693]} value -> { }
    Type of selection= { }
    Templates called by selection (2)
      {bigInt} * {nil} -> { }
      {smallInt} * {nil} -> { }
  primeNumbers
iteration
printing
```

Figure 71. Locating a potential error with the type-based browser

The numbers refer to the explanation given in the main text.

itself, thus `nil` could not have originated there. The `reversePower:` template, however, may be interesting. The programmer selects it.

6. In response, the system opens a browser on the `reversePower:` method. Here, finally, the reason for the unsafe send can be found: the local slot `factor` has the initial value `nil` (the default initial value in Self). Moreover, the structure of the code manipulating `factor`, a three-way comparison in which two of the argument blocks assign to `factor` and the third performs a non-local return (if executed), defeats the type inferencer's flow-sensitive analysis. Thus, the inferred type of the send reading `factor` includes $\overline{\text{nil}}$; even though the initializer in this case is dead. Thus, the application will in fact never attempt to multiply a bigInt with `nil`, but the type inferencer and checker could not eliminate the possibility. To prevent the checker from raising this unnecessary alert again in the future, the programmer could initialize the `factor` slot to a smallInt effectively eliminating $\overline{\text{nil}}$ from its type. This change

would also make `reversePower:` easier to understand by giving the reader a hint that the `factor` slot contains an integer.

7.2.1 Discussion

When a type browser is opened on a method, it is given a set of templates for the method. Often, the browser receives exactly one template for the method, but in general, it can accept a set of templates. For all the type-based operations, the browser consults this set of templates. For example, to display the type of an expression, the browser computes the union of the types of the expression in each of the templates; to find all methods that a send may invoke, the browser collects targets for the send in each of the templates; and to find all sends that may invoke the method, the browser searches the type inferencer's collection of templates for the current application, looking for any that contain a send connected to one of the templates in the browser's set of templates.

Since a type browser is associated with a set of templates, it is possible to open multiple type browsers for different sets of templates on a single method. For example, the bigInt multiplication method can be shown in one browser using the template $\{\overline{\text{bigInt}}\}\ \texttt{*}\ \{\overline{\text{bigInt}}\} \rightarrow \{\overline{\text{bigInt}}\}$, and in another using $\{\overline{\text{bigInt}}\}\ \texttt{*}\ \{\overline{\text{nil}}\} \rightarrow \{\}$. It was this feature that allowed the type browser in the example above to locate all sends that call the bigInt multiplication method with `nil` as the argument and ignore those that supply reasonable arguments such as bigInts or smallInts. In contrast to the type browser, the Self programming environment avoids opening two outliners on the same object to support the principle of direct and viewless interaction with objects. More work will be needed to resolve these contrasting approaches.

The Self 4.0 programming environment tools and the type-based tools also differ in their scope. The former tool set operate on the image in its entirety: requesting senders of, say, "`*`" yields a list of all methods that send "`*`" in the entire image. In contrast, the type browser's equivalent of senders, finding the templates invoking a given template, searches *only* among the templates of the given application. Clearly, the programmer using these tools must be aware of this difference in scope.

While our experience with the type-based browser is still limited, our preliminary results are positive. We have found that the integration with the checker and the use of colors effectively draws attention to places in a program where errors may happen. Moreover, the browser's navigational features offers the programmer a more direct alternative than the senders and implementors tools for navigating through programs.

7.3 Optimizing compilation

Traditionally, object-oriented languages, and in particular dynamically-typed languages, have been less efficient than procedural languages. Recent work by several researchers has significantly reduced this deficiency. Most notable, perhaps, are the dynamic compilation techniques pioneered for Smalltalk by Deutsch and Schiffman [41] and later adopted by Chambers [26] and Hölzle [58] in the Self system. Type inference, since the early work in Smalltalk by Suzuki (see Section 2.3.2.3), has been pursued as a means to optimize programs using more traditional techniques than the dynamic compilation ideas. However, many of the early type-inference systems were either not precise enough to support effective optimizations, inferred abstract types which are not so useful for a compiler, or could not cope with realistic programs. While our type inferencer still has room for improvements, it performs at a level where type inference's applicability to optimizing compilation can be assessed. Together with Urs Hölzle, we performed such a study. The main results are summarized below; more details can be found in [5]. Recently, other researchers have performed similar studies: Grove applied the basic type inference algorithm to optimizing Cecil [51], and Plevyak and Chien applied the iterative algorithm to optimizing Concurrent Aggregates [95, 96].

Compared with procedural programs, object-oriented programs complicate optimizing compilation by their use of dynamically-dispatched sends. For example, consider a send expression like

```
    point1 + point2.
```

If the compiler has no information about the receiver of the "`+`" message, it must generate code sufficiently general to handle all possible cases, including integers, points, rectangles and other kinds of objects that understand "`+`". Effectively, the compiler must generate code that performs the dynamic dispatch at run time. In contrast, if the compiler has access to type information that states that the expression `point1` has type {$\overline{\text{cartesianPoint}}$}, the compiler can replace the expensive dynamic dispatch with a direct (and fast) call to $\overline{\text{the point “+” method}}$ for cartesian points, or even inline this method. If the available type is less precise, such as {$\overline{\text{cartesianPoint}}$, $\overline{\text{rectangle}}$}, the compiler can still generate better code than if it knows nothing about the receiver expression. However, with less precise type information, lower quality code should be expected.

The standard Self compiler collects type information dynamically by monitoring an instrumented version of the application. This technique, developed by Hölzle and known as type feedback (see Section 2.3.2.12 for a brief review), has been shown to provide excellent performance, significantly outperforming commercial Smalltalk systems [59][†].

To quantify how well type inference can support optimization, we cloned and modified the standard Self compiler to use type inference instead of type feedback; see Figure 72. Thus, our experiment involved two compilers that are identical except for their source of type information: one uses type feedback, the other type inference. We compiled 23 benchmark programs, ranging in size from 3 to 7,000 lines (not counting standard library code; up to 11,000 lines when counting library code), with each of the two compilers. Finally, we compared the performance of the code generated by the compilers. Evaluating type inference by comparing it against the standard set by type feedback focuses on the effect of the available type information, and reduces the significance of other factors that influence absolute performance: compiler back-end, programming language, programming style, etc.

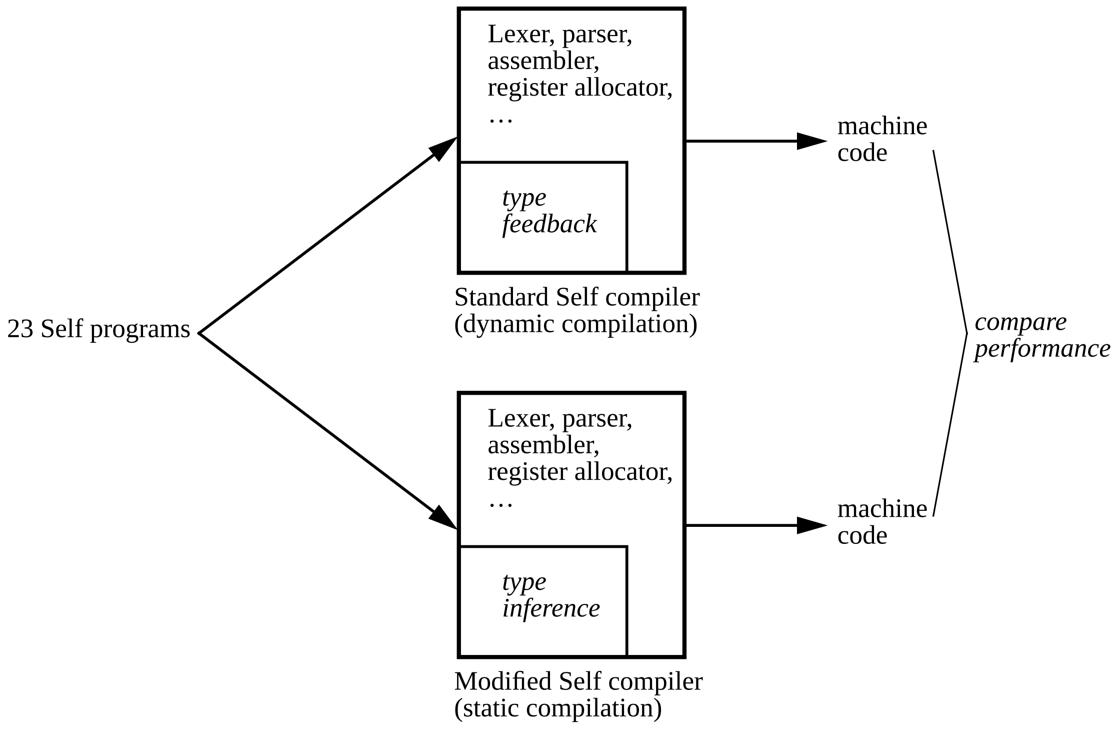

23 Self programs
Lexer, parser, assembler, register allocator, …
type feedback
Standard Self compiler (dynamic compilation)
Lexer, parser, assembler, register allocator, …
type inference
Modified Self compiler (static compilation)
machine code
machine code
compare performance

Figure 72. The setup for evaluating type inference for optimization

In the performance data presented for the benchmarks below, we summarize the smallest four benchmarks, whose sizes range from 3 to 10 lines of source code, under the label “tiny,” and twelve slightly larger benchmarks, whose

[†] Compiler performance can mean compilation time, size of generated code, speed of generated code, etc. While all these aspects are relevant, the present study emphasized execution speed of the generated code.

sizes range from 20 to 170 lines of source code, under the label “small.” The larger benchmarks are presented individually. Most of them are the same as the ones used throughout this dissertation. The additional benchmarks used in the compilation study are summarized in Table 30 using the same format as in Table 7 on page 65.

Benchmark	Appl. size[a]	Total size[b]	File size[c]	Description
SelfParser	400	1,443	2,711	Handwritten recursive decent parser for the 1989 version of Self [26]
CParser	7,000	10,941	71,231[d]	Lexer, LR(1) parser, and tree builder for ANSI C; generated by Mango [1]
RSA	300	1,541	2,740	Probabilistic primality checking and public-key encryption and decryption using the RSA algorithm [101] (tests bigInts)

Table 30. Additional large benchmark programs used in compilation study

a. Approximate lines of code in method bodies (i.e., expressions), excluding “standard code” such as integers, lists, etc. The line counts exclude blank lines.
b. Number of lines in the bodies of the methods that our application extractor (see Chapter 6) deems are part of the application; includes methods in standard objects such as integers, lists, and booleans.
c. Number of lines in the source file produced when the application extractor is applied to the application; these counts, unlike “Appl. size” and “Total size,” include blank lines and definitions of instance variable, parents slots, etc.
d. The CParser application needs a significant number of “data objects” to form the parse tables. These increase the “File size,” but contribute little code to “Appl. size” and “Total size.”

Since we have observed in our work how Self’s automatic coercion of smallInts into bigInts upon arithmetic overflow confounds the type inferencer (lacking range analysis being the primary shortcoming), we compiled the 23 benchmark programs twice with the type-inference-based compiler. The first compilation permitted the smallInt to bigInt coercion; the second blocked it by treating smallInt overflows as an error (similar to how Eiffel, say, treats overflows). To illustrate the difference between the two cases, consider the type inferred for `3+4`. In the former case the type is the conservative {$\overline{\text{smallInt}}$, $\overline{\text{bigInt}}$}; in the latter case it is the precise {$\overline{\text{smallInt}}$}. Shivers, in his work on type inference for Scheme, also encountered this problem of being unable to avoid pollution of types by $\overline{\text{bigInt}}$ [107]. He states:

> From the optimising compiler’s point of view, the biggest piece of bad news in the Scheme type system is the presence of arbitrary-precision integers or “bignums.”

Figure 73 shows the run-time for the benchmarks programs in each of the three cases (TF for type feedback, TI for type inference, and TI-int for type inference without bigInts). Overall, the fastest system is TI-int, followed by TF, and then TI (although TI outperforms TF on two of the seven large benchmarks). With the exception of the small benchmarks for which TI performed badly, all three systems deliver quite similar performance, the relative difference in run time being typically less than 15%. The reason why TI performs badly on the small benchmarks is subtle. The TI system conservatively predicts that bigInts may occur in several frequently executed expressions, including loop counters. In contrast, TI-int has been hardwired to exclude bigInts, and TF avoids them since arithmetic rarely overflows in practice. While the immediate effect of the conservatism of TI would be small, a secondary effect aggravates the consequences: the prediction of bigInts interacts with the compiler’s splitting heuristics, preventing it from splitting crucial control-flow paths and in turn causing loss of further optimization opportunities. A detailed discussion of this problem and a concrete example can be found in [5]. Fortunately, on the larger benchmarks, which are less dominated by integer arithmetic, the three systems deliver quite similar performance.

Although our study comprises only a limited set of benchmarks and focuses on Self, the comparable and high performance delivered by type feedback and type inference indicates that both optimization techniques will play a role in the future. In particular, since the techniques differ significantly in other regards than run-time performance, the broader context may decide which technique to favor on a case-by-case basis. For example, type feedback, working on a method at a time, is well suited for program development where the programmer wants interactiveness (short or even no compilation pauses), yet fast execution. However, type feedback cannot straightforwardly compile “standalone” versions of applications. In contrast, type inference, being performed statically, allows the construction of stand-alone applications through extraction and static compilation, but may have problems supporting some of the

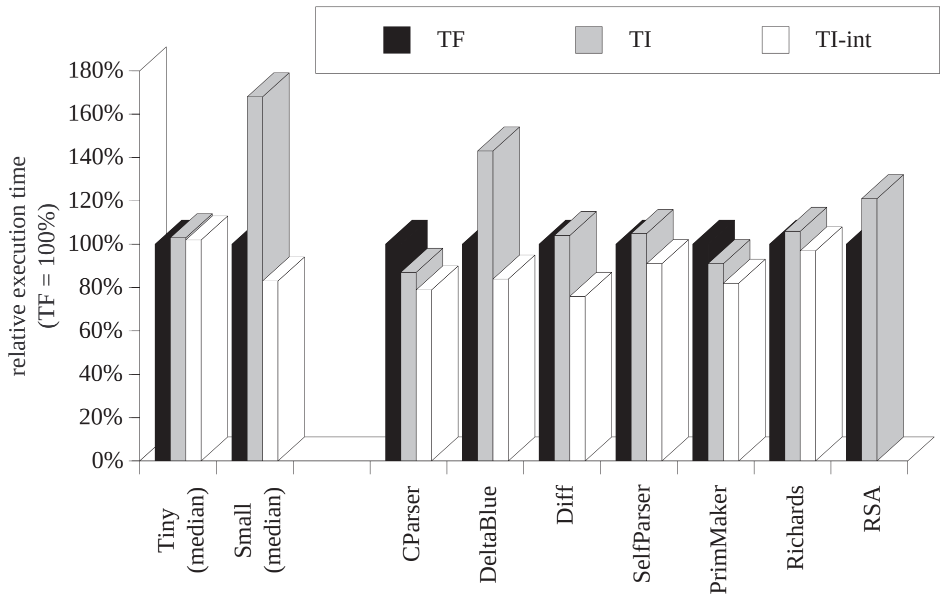


Figure 73. Relative execution time of the benchmarks

The three systems shown are TF (type feedback), TI (type inference), and TI-int (type inference without bigInt arithmetic). Since the RSA benchmark relies on bigInts, it would fail under TI-int.

extremely dynamic features of the Self environment, e.g., reflection or on-the-fly code creation. A thorough comparison of the two techniques involves several more issues; see [5].

Furthermore, our study suggests that type inference eliminates the performance penalty of dynamic typing compared to static typing for delivered applications. Hölzle observes that when C++ programs are written in a style that matches the degree of object-orientation found in Self programs, the C++ type-declaration-based compiler produces no better results than Self's type-feedback-based compiler [58] (Section 7.3.1). Since our type-inference-based system matches the performance of type feedback while providing for the delivery of statically-compiled applications, we can now assert that dynamic typing per se need no longer detract from the performance of delivered applications.

Finally, the call graph that the type inferencer builds enables data-flow analysis of object-oriented programs, bringing them within range of most of the global optimization techniques developed for procedural programs over the last two decades.

7.4 Summary

In our pursuit for supporting delivery of applications from dynamically-typed object-oriented environments, we have presented three more programming tools made possible by type inference: a static checker, a browser, and an optimizing compiler:

- *Static checker.* Type inference can support static detection of certain kinds of errors, including message-not-understood, multiple inheritance ambiguities, and non-LIFO block invocations. With Ole Lehrmann Madsen, we have implemented a prototype checker tool that uses the inferred types to detect sends that may cause message-not-understood errors. The checker classifies sends as green (those that will never fail), amber (those that may fail), or red (those that will always fail, if executed). Just as with a conventional declaration-based checker for a statically-typed language, the inference-based checker assures that any send that may cause an error will be caught. Unlike conventional checkers, the Self checker does not prevent the execution of programs with potential errors.

- *Browser.* Programmers often need to navigate through code, be it to understand how a set of methods perform a given task or to get from the place where an error was detected (by the static checker or at run time) to where it was caused. Dynamically-dispatched message sends make this navigation harder since it is not manifest which method a given message invokes. Present text-based tools, including “senders” and “implementors” in Smalltalk, lack precision when dealing with common message selectors. A type-inference-based browser, like the one we have built in collaboration with Ole Lehrmann Madsen, can avoid this problem. Using type information, this browser can precisely navigate forward from a dynamically-dispatched send to the methods it may invoke, and backwards from a method to all the sends that may invoke it.

- *Optimizing compiler.* Type inference has been considered an important component in optimizing object-oriented languages. Collaborating with Urs Hölzle, we performed an experiment to evaluate this belief. We built two Self compilers, identical except for their source of type information. One compiler uses type inference, the other type feedback, which in previous work has been shown to deliver state-of-the-art performance. Comparing execution times of a suite of 23 Self programs translated with each compiler, we found only small differences between type inference and type feedback. Thus, with type inference, dynamic typing per se need no longer detract from the performance of delivered applications, even if one wants to compile statically for delivery.

In conclusion, using the suite of tools described in this and the previous chapter, dynamically-typed object-oriented environments can deliver safe, efficient, and compact applications.

8 Conclusions

Modern object-oriented programming environments demand concrete type information for the delivery of applications: compilers need it to optimize programs, browsers need it to assist programmers in navigating through code, and extractors need it to wrap up compact applications. But concrete type information cannot be obtained from type declarations without crippling a fundamental feature of object-oriented languages: polymorphism. Hence, the ability to infer types becomes highly desirable. Unfortunately, while polymorphism creates the need for type inference, it also turns out to be the hardest obstacle for type inference. Many early attempts, although contributing to the understanding of the problem, were not completely successful.

We introduced the notion of templates in order to uniformly characterize existing constraint-based algorithms for inferring concrete types of code with parametric polymorphism. The high-level descriptions made possible by templates allowed direct comparisons of the algorithms and shed light on their strengths and weaknesses. In particular, templates helped us understand why parametric polymorphism leads to precision loss for the earlier type-inference algorithms: they cannot maintain sufficient separation of differently-typed method invocations.

Based on the insight obtained by studying the properties of the previous algorithms, we designed and implemented the cartesian product algorithm (CPA), a polyvariant adaptive type-inference algorithm for analyzing code with parametric polymorphism. CPA offers several advantages:

- *Precision*. By computing cartesian products of actual argument types CPA reduces the analysis of each polymorphic send to a monomorphic case analysis, ensuring better precision than previous algorithms. Measurements show that CPA achieves the smallest average size of the inferred types.
- *Efficiency.* Monotonicity of the cartesian product allows an integrated (one-pass) algorithm while still avoiding the redundancies of the non-adaptive expansion algorithms. Measurements show that CPA outperforms even the monovariant basic algorithm on larger benchmarks.
- *Generality.* Unlike other algorithms that expand away inheritance, CPA handles it directly, permitting precise analysis of a wider range of languages, including those with inheritance of state, dynamic inheritance, multiple inheritance, and multiple dispatch.
- *Simplicity.* CPA was motivated by the observation that at run time send sites may be polymorphic, but a particular execution of a send is always monomorphic. Simulating this behavior accurately, CPA needs no code expansions, no inheritance elimination, and no iteration, resulting in a conceptually simple algorithm.

The cartesian product algorithm addresses parametric polymorphism. It can be combined with algorithms that other researchers have developed for analyzing code with data polymorphism. A complete type inference system must address other features as well. This dissertation offers a range of techniques for dealing with several central features of object-oriented languages: inheritance (including multiple, dynamic, and of state), dynamic dispatch (possibly on multiple arguments), lexically-scoped blocks, non-local returns, and flow-sensitive type inference for variable accesses. In addition, we developed grouping to analyze objects directly, and an incremental analysis technique to speed up repeated analysis of versions of the same program or sequences of similar programs.

The cartesian product algorithm, combined with these techniques, demonstrates that concrete type inference for object-oriented programs is *possible*. To demonstrate that concrete type inference is *useful* for the delivery of applications and to verify that our inferencer achieves sufficient precision to drive high performance type-based tools, we prototyped several such tools, including an extractor, a checker, a browser, and a compiler.

The *extractor* constitutes our most complete type-based tool. It enables, for the first time, delivery of compact applications from integrated dynamically-typed object-oriented environments while achieving soundness (full behavior preservation across extraction), automation (no programmer involvement required), efficiency (extraction takes only minutes), effectiveness (large size-reductions), and generality (no reliance on specific properties of the Self world). Powerful extraction technology, as enabled by concrete type inference, can significantly reduce or even eliminate the "final application size" concern that has been held against Smalltalk-like programming environments.

The static *checker* and *browser*[†], integrated in our implementation, use type information to color send expressions according to whether they never fail, may fail, or always fail (if executed). Like a conventional declaration-based type checker for a statically-typed language, the inference-based checker assures that any send that may cause a run time message-not-understood error will be caught. The browser presents code in the colors computed by the checker and uses types to trace flow of control and thereby data flow. The tracing can help programmers both in debugging their own code and in understanding other programmers' code that they are about to reuse.

The type-inference-based *compiler*[‡], identical to the standard Self compiler except for its source of type information, achieves compiled-code performance comparable to the standard—and state-of-the-art—dynamic Self compiler. Since the type-inference-based compiler employs no run-time information, it can be used for "traditional" static compilation, yielding stand-alone applications that need no compiler for execution (assuming that the application uses none of the reflective features such as run-time generation of new kinds of objects).

Thus, with precise concrete type inference, applications can be statically checked (to ensure the absence of certain kinds of errors), extracted (to improve compactness), compiled (to improve efficiency), and shipped. In other words, type inference finally enables delivery of applications from dynamically-typed object-oriented environments.

8.1 Applicability

We performed our work within the context of the dynamically-typed integrated Self system, but our results have broader applicability: statically-typed languages and non-integrated environments need concrete type information too.

Most statically-typed languages, including Beta, C++, and Eiffel, employ class-types. Since class-types are not fully concrete and since programmers cannot use maximally specific type declarations without making their code less reusable, type declarations do not allow as powerful type-based tools as do concrete types. For example, a compiler using the declared class-types of receiver expressions can inline less than a compiler using concrete types. Thus, even when programs contain static class-type declarations, concrete type inference can often compute more specific type information.

Other statically-typed languages, including TOOPL and Strongtalk, do not use class-types. Instead, they separate classes and types to increase polymorphism: unlike languages with class-types, the abstract type systems of TOOPL and Strongtalk permit substitutability of instances of classes that are not related by subclassing. Consequently, the static type declarations in TOOPL and Strongtalk provide even less information about concrete types than do class-types, and therefore are also less useful than class-types when implementing tools that require concrete types. In general, it seems that the more flexible static type systems become, the more important concrete type inference becomes.

The performance conscious C++ community in particular should consider inference of concrete types. Concrete types provide the information necessary to statically resolve dynamic dispatch, thereby opening up C++ (and any other object-oriented language) to the whole range of traditional data-flow-based optimization techniques. Without the full use of these techniques, performance of C++ cannot reach the same level as performance of the monomorphic non-object-oriented C language.

Non-integrated environments, as are typical for C++ today, need no extraction, but still suffer from dead code in the applications. While perhaps at a tolerable level presently, the amount of dead code will likely increase in the future as reusable class libraries and, on a larger scale, frameworks become widespread. Improved operating-system support for sharing of objects may alleviate this problem in the longer term. In the meantime, extraction technology could be applied during the linking phase to eliminate dead code from delivered applications.

Finally, while abstract static type systems help catch some errors such as message-not-understood, other classes of errors cannot be detected effectively based on abstract types. For instance, the detection of potential or certain

[†] Implemented in cooperation with Ole Lehrmann Madsen.

[‡] Implemented in cooperation with Urs Hölzle.

accesses to uninitialized variables relies inherently on the control-flow information provided by concrete types. Thus, even in statically-typed languages, more errors can be detected statically with concrete type inference than without.

8.2 Future work

Throughout the previous chapters, we have mentioned shortcomings and possible improvements of the techniques being discussed. Below we summarize the most important areas for future work.

Data polymorphism. The cartesian product algorithm addresses parametric polymorphism only. The other side of the coin, data polymorphism, cannot be ignored. While it has not been a focus of our work, other researchers have developed algorithms for data polymorphism, the most powerful being the data-iterative algorithm, which we discussed in Section 4.3.2. Future work includes combining the cartesian product algorithm and the data-iterative algorithm. A more ambitious project would be the development of an integrated (non-iterative) algorithm for analyzing data polymorphism. This algorithm would have the potential to improve performance because it avoids iteration overhead.

Scaling. The question of how well type inference scales to very large programs merits further investigation. The largest program that we have successfully inferred types for is CParser, consisting of approximately 11,000 lines of code in method bodies (i.e., not counting slot declarations, formal arguments, and so on). Attempting to apply the type inferencer to even larger programs has not been successful, mainly for two reasons: data polymorphism and encountering unsupported (reflective) features. First, our inferencer's lacking precision on code with data polymorphism hurts it comparatively more on larger programs: the larger the target program, the more likely it uses containers of several different kinds of objects. Second, finding large Self programs that do not use reflection can be surprisingly hard, as discussed in Section 6.3.2.2. Because of these two limitations of the implemented type inferencer, we have not been able settle the scaling issue yet. A priori, though, we expect the cartesian product algorithm to scale well. Unlike expansion algorithms that customize per call-site, the cartesian product algorithm customizes according to *inferred* argument types. This difference may be crucial for scaling:

- The number of call sites for certain methods (such as the arithmetic methods) grows linearly with program size, disfavoring the expansion algorithms since they reanalyze the methods for each call size.
- The number of argument types for most methods (although not `ifTrue:False:` and other methods implementing standard control structures) grows sublinearly with program size, favoring the cartesian product algorithm, since it only reanalyzes methods when it encounters new types of arguments.

Modularity. Separate compilation, or more generally modular analysis, as provided by Milner's type inference system has obvious advantages: analysis need not scale to the size of whole programs but only to the size of modules, and program changes only invalidate analysis results within a single module. While concrete type inference inherently seems to be a whole-program analysis, investigating whether some of the advantages of modularity can be transferred would be worthwhile.

Range analysis. Self's transparent coercions of smallInts to bigInts upon overflow contributes significantly to the complexity of type inference, since even the smallest program is likely to use integer arithmetic. The presence of bigInts in the inferred types is unfortunate, since in most "practical" situations arithmetic does not overflow. For example, in the Diff file comparison application integer arithmetic will not overflow unless extremely large files with more than 2^{30} bytes are compared (and even then, the Diff program would probably run out of memory before hitting the overflow point). Worse still, if the primary application of type information is optimizing compilation, the type inferencer's inability to rule out overflows consistently injects bigInts into the inferred types of the most critical expressions in the program: the loop counters. Hence, combining type inference with an effective range analysis would allow better precision, and at the same time make type inference more efficient since the overflow code and the bigInt implementation need not be analyzed unnecessarily.

Integration in the programming environment. Although we have implemented several type-based tools, these have been proof-of-concept only. A complete integration of the tools in the Self programming environment is still missing, and would involve work on several interesting interfaces, including:

- Integration of the type inferencer and the debugger to be able to compute and present more specific type information, taking into account that a certain error has just been observed.
- Integration of the type inferencer and the editor to allow selective invalidation and recomputation of inferred types after program changes.

Reflection and other features. Finally, although type inference handles a large part of the Self language, there are features that it does not support or supports only partially, reflection being the most noticeable. Extending type inference to reason about reflective operations such as adding slots to or removing slots from objects seems plausible. One problem that currently prevents us from supporting such operations is the lack of information about the affected slots. The reflective primitives in the Self system use strings (names) to specify which slot they manipulate. Thus, unless the name argument is known more precisely than $\overline{\text{string}}$ (i.e., its value is known, not just its object type), the type inferencer must treat the primitive very conservatively. It must assume that any slot of the manipulated object may be affected. With some form of constant propagation and constant folding to narrow down the arguments specifying slot names, better support for reflection would be possible. The current shortcomings when analyzing “performs” (see Section 6.3.2.2) would also be alleviated by more precise information about the performed string values. Of course, in general such information may be impossible to infer, since it can be data read from files or otherwise be unavailable at inference time. Therefore, while more powerful analysis techniques can eliminate some of the shortcomings of the currently implemented type inference system, in a language as expressive as Self, some features will probably still present obstacles and may require limited use of declarations as a backup strategy.

8.3 Summary

Concrete type inference has been the Holy Grail of object-oriented language implementation: wanted for the benefits it would bring to delivery of applications, but difficult to reach exactly for the reason that it was needed—polymorphism. This dissertation contributes an understanding of the inherent difficulty of inferring types of polymorphic code, shows that type inference is possible using the cartesian product algorithm, and demonstrates that type inference enables delivery of safe, compact, and efficient applications through static checking, browsing, extraction, and optimizing compilation. Indeed, type inference supports these tasks better than the type declarations commonly found in statically-typed object-oriented languages, be they based on classes or more recent ideas of separating classes and types.

Ultimately, we hope that a successful system for inferring concrete types will liberate both the language designer and the practicing programmer. The language designer can now pursue his goals without having to balance expressiveness and performance of the environment: the designer can opt for an expressive and abstract type system (or *no* static type system), knowing that this goal does not debilitate the delivery tools that need concrete type information. The programmer can choose a language fit for expressing a solution to the problem at hand, instead of being concerned with whether a candidate language allows programs to execute sufficiently fast or fit within the memory available. Concrete type inference reconciles expressiveness and application delivery.

Glossary

Abstract type. An abstract type specifies behavior or properties of objects, usually externally observable properties such as their interface, i.e., the operations they support (see Section 2.1.1). An abstract type can be implemented by more than one concrete type.

Adaptive. An adaptive type-inference algorithm uses type information during type inference to improve the precision/efficiency trade-off (see Section 3.2.5).

BigInt. An integer representation that supports arbitrarily large values (e.g., not limited to 32 bits). BigInts are implemented in Self code, not in the virtual machine.

Block. Blocks in Self correspond to lambda expressions in Scheme. They evaluate to closures. See Section 4.4 for a description of their behavior.

Clone family. An initial object and all objects cloned directly or indirectly from it (see Section 3.1.2).

Closure. Self's closures correspond to Scheme's. They contain a reference to a block and to the activation record in which the block was evaluated to produce the closure. See Section 4.4 for details.

Concrete type. A concrete type specifies the implementation of objects: which instance variables they have, how they are laid out in memory, and the code they execute to perform operations (see Section 2.1.1). In class-based languages, the concrete type of an object is the exact class it was instantiated from; in the classless Self language, *object types* substitute for classes (see Section 3.1.2). Expressions (variables) have *sets* of object types as their concrete type, since they can evaluate to (hold) objects of multiple object types (see Section 3.1.2).

Data polymorphism. The ability of a slot (instance variable) to hold objects with multiple object types. For example, data polymorphism occurs when a program builds lists of integers and lists of booleans.

Delivery. The construction of time-efficient, type-safe, and compact executable versions of applications suitable for distribution to users.

Dispatch resolution. The process of statically determining a conservative approximation to the set of methods a dynamically-dispatched send may invoke (see Section 4.1).

Dynamic dispatch. The run-time selection, based on the receiver object, of which method to invoke for a given message.

Flow (in)sensitive. A flow-sensitive type-inference algorithm accounts for the sequencing of assignments and read-accesses to variables to infer more precise types for variable accesses; a flow-insensitive algorithm ignores this sequencing (see Section 4.5).

Group. A finite set of structurally similar initial objects. For example, all smallInts form a group, and all point objects form a group (see Section 3.1.2 and Section 4.6).

Initial object. An object that was created prior to the execution of the application. It can play a role like classes in class-based languages (a "prototype" object that the application can clone), but may also define data used by the application (see Section 3.1.2).

Message. A message results from the evaluation of a send expression. The message carries a request to the receiver object to perform some operation. The method implementing the operation is determined by a dynamic dispatch.

Monovariant. A monovariant type-inference algorithm analyzes each method exactly once: it creates one template per method (see Section 3.2.2).

Object type. The closure under cloning of a group, i.e., the union of the clone families of the initial objects in the group. Corresponds to classes in a class-based language. Every object has an object type (see Section 3.1.2).

Parametric polymorphism. The ability of a method to be invoked on arguments of multiple object types. For example, the `max:` method in Self works on both smallInts, bigInts, and strings.

Polymorphism. The ability of code to work on objects of any object types as long as they satisfy the abstract type required by the code (see Section 2.1.3).

Polyvariant. A polyvariant type-inference algorithm may analyze each method multiple times: it may create more than one template for some methods (see Section 3.2.2).

Recursive customization. Out-of-control type inference (or compilation); a positive feedback loop necessitates reinferring types for (or recompiling) a method as soon as type inference (compilation) of the method finishes (see Chapter 5).

Send or **send expression**. A syntactic (static) entity that can be executed, resulting in a message. Sends resemble procedure calls in non-object-oriented languages, but do not have a fixed target: their execution involves a dynamic dispatch.

SmallInt. An integer representation with a limited range (30 bits in Self 4.0). This representation is defined by the virtual machine.

Template. A representation of the control and data flow within a method, along with types of all arguments, the result, local slots, and expressions. Templates enable a concise and uniform description of type-inference algorithms for analyzing code with parametric polymorphism.

References

1. Agesen, O. *Mango—A Parser Generator for Self*. Sun Microsystems Laboratories Technical Report, SMLI TR-94-27, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, June 1994.
2. Agesen, O. Constraint-Based Type Inference and Parametric Polymorphism. In *SAS'94, First International Static Analysis Symposium,* p. 78-100, Namur, Belgium, September 1994. Springer-Verlag (LNCS 864).
3. Agesen, O. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming,* p. 2-26, Århus, Denmark, August 1995. Springer-Verlag (LNCS 952).
4. Agesen, O., L. Bak, C. Chambers, B.W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, M. Wolczko. *The Self 4.0 Programmer's Reference Manual*. Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043, USA, June 1995.
5. Agesen, O. and U. Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications,* p. 91-107, Austin, Texas, October 1995.
6. Agesen, O., J. Palsberg, and M.I. Schwartzbach, Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP'93, Seventh European Conference on Object-Oriented Programming,* p. 247-267, Kaiserslautern, Germany, August 1993. Springer-Verlag (LNCS 707). An extended version of this paper is published in: *Software—Practice & Experience*, 25(9), p. 975-996, September 1995.
7. Agesen, O. and D. Ungar, Sifting Out the Gold: Delivering Compact Applications from an Object-Oriented Exploratory Programming Environment. In *OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications,* p. 355-370, Portland, Oregon, October 1994.
8. Assumpção Jr., J.M. de and S.T. Kufuji. Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection. *ECOOP'95 Workshop: Advances in Metaobject Protocols and Reflection,* Århus, Denmark, August 1995.
9. Aho, A.V, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, Massachusetts, 1986.
10. Aiken, A. E.L. Wimmers, and T.K. Lakshman. Soft Typing with Conditional Types. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* p. 163-173, Portland, Oregon, January 1994.
11. Apple Computer Eastern Research and Technology. *Dylan: an Object-Oriented Dynamic Language*. Apple Computer, 1 Main Street, Cambridge, Massachusetts 02141, April 1992.
12. Barnard, A.J. *From Types to Dataflow: Code Analysis for an Object-Oriented Language*, Ph.D. thesis, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, October 1992.
13. Black, A., N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 78-86, Portland, Oregon, September 1986.
14. Bobrow, D.G., L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. Common Lisp Object System, *SIGPLAN Notices*, 23(1), special issue, p. 1-48, September 1988.
15. Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 17-29, Portland, Oregon, September 1986.
16. Boreczky, J. and L. Rowe. Building Common Lisp Applications with Reasonable Performance. In *Proceedings of the Third International Lisp Users and Vendors Conference*, Cambridge, Massachusetts, August 1993. Published in *SIGPLAN Lisp Pointers*, 6(3), p. 20-29, July-September 1993.
17. Borning, A.H. Classes versus Prototypes in Object-Oriented Languages, *ACM/IEEE Fall Joint Computer Conference*, p. 36-40, Dallas, Texas, November 1986.
18. Borning, A.H. and D.H.H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, p. 133-141, Albuquerque, New Mexico, January 1982.
19. Bracha, G. and D. Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *OOPSLA'93 Object-Oriented Programming Systems, Languages and Applications,* p. 215-230, Washington, DC, September 1993.
20. Bruce, K.B. Safe Type Checking in a Statically-Typed Object-Oriented Programming Language. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 285-298, Charleston, South Carolina, January 1993.

21. Cardelli, L. *Basic Polymorphic Typechecking*, Computing Science Technical Report No. 112, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, September 1984.

22. Cardelli, L. A Semantics of Multiple Inheritance. *Information and Computation*, 76(2-3), p. 138-164, February-March 1988. A previous version can be found in Proc. *International Symposium on Semantics of Data Types*, p. 51-67, Sophia-Antipolis, France, June 1984. Springer-Verlag (LNCS 173).

23. Cardelli, L. and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), p. 471-522, December 1985.

24. Cartwright, R. and M. Fagan. Soft Typing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, p. 278-292, Toronto, Ontario, Canada, June 1991.

25. Castagna, G. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3), p. 431-447, May 1995.

26. Chambers, C. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California, October 1991.

27. Chambers, C. Object-Oriented Multi-Methods in Cecil. In *ECOOP'92, Sixth European Conference on Object-Oriented Programming*, p. 33-56, Utrecht, The Netherlands, June 1992, Springer-Verlag (LNCS 615).

28. Chambers, C., D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June 1991. Originally published in *OOPSLA'89 Object-Oriented Programming Systems, Languages and Applications*, p. 49-70, New Orleans, Louisiana, October 1989.

29. Chase, D.R., M. Wegman, and F.K. Zadeck. Analysis of Pointers and Structures. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, p. 296-310, White Plains, New York, June 1990.

30. Chien, A. A., Karamcheti, V., and Plevyak, J. *The Concert System - Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs*. Technical Report No. UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois Urbana-Champaign, June 1993.

31. Choi, J.-D., M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 232-245, Charleston, South Carolina, January 1993.

32. Consel, C., Polyvariant Binding-Time Analysis For Applicative Languages. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'93*, p. 66-77, Copenhagen, Denmark, June 1993.

33. Cook, W.R. A Proposal for Making Eiffel Type-safe. In *ECOOP'89, Third European Conference on Object-Oriented Programming*, p. 57-70, Nottingham, U.K., July 1989, Cambridge University Press.

34. Cooper, K.D., M.W. Hall, and K. Kennedy. Procedure Cloning. In *Proceedings of the 1992 International Conference on Computer Languages*, p. 96-105, Oakland, California, April 1992.

35. Cousot, P. and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, p. 238-252, January 1977.

36. Curtis, P. *Constrained Quantification in Polymorphic Type Analysis.* Ph.D. thesis, Department of Computer Science, Cornell University, 1990. Also available as Technical Report CSL-90-1, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, California 94304, February 1990.

37. Cytron, R., J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4), p. 451-490, October 1991.

38. Dahl, O.J., B. Myrhaug, K. Nygaard. *Simula 67 Common Base Language.* Norwegian Computing Center, Oslo, Norway, 1968.

39. Damas, L. and R. Milner. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, p. 207-211, Albuquerque, New Mexico, January 1982.

40. Deutsch, L.P. The Past, Present, and Future of Smalltalk. In *ECOOP'89, Third European Conference on Object-Oriented Programming*, p. 73-87, Nottingham, U.K., July 1989, Cambridge University Press.

41. Deutsch, L.P. and A.M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, p. 297-302, Salt Lake City, Utah, January 1984.

42. Digitalk Inc. *Smalltalk/V for Win32 Programming, Reference Manual,* Chapter 17: Object Libraries and Library Builder, 1993.

43. Fosdick, L.D. and L.J. Osterweil. Data Flow Analysis in Software Reliability. *Computing Surveys,* 8(3), p. 305-330, September 1976.

44. Fraser, C.W. and A.L. Wendt. Integrating Code Generation and Optimization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, p. 242-248, Palo Alto, California, June 1986.

45. Freudenberger, S.F. and J.C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Proceedings of the International Workshop: Code Generation - Concepts, Tools, Techniques.* p. 146-170, Dagstuhl, Germany, May 1991.

46. Goldberg, A. and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

47. Goodman, J.R. and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Conference Proceedings of the 1988 International Conference on Supercomputing,* p. 442-452, St. Malo, France, July 1988.

48. Graver, J.O. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989.

49. Graver, J.O. and R.E. Johnson. A Type System for Smalltalk. In *Conference Record of the Seventeenth Symposium on Principles of Programming Languages*, p. 136-150, San Francisco, California, January 1990.

50. Greve, J. and J.C. Olsen. *Type Inference of BETA*, Master's Thesis, Computer Science Department, Aarhus University, Ny Munkegade, Århus, Denmark, May 1995.

51. Grove, D. The Impact of Interprocedural Class Inference on Optimization. In *Proceedings* of *CASCON'95 Centre for Advanced Studies Conference,* p. 195-203, Toronto, Ontario, Canada, November 1995.

52. Harrold, M.J. and M.L. Soffa. Efficient Computation of Interprocedural Definition–Use Chains. *ACM Transactions on Programming Languages and Systems,* 16(2), p. 175-204, March 1994.

53. Hense, A.V., *Polymorphic Type Inference for a Simple Object Oriented Programming Language With State*. Tech. Bericht Nr. A 20/90 (Technical Report), Universität des Saarlandes, 1990.

54. Hense, A.V. and G. Smolka. *Principal Types for Object-Oriented Languages*, Technischer Bericht Nr. A 02/93, Universität des Saarlandes, Im Stadtwald 15, 6600 Saarbrücken 11, Germany, June 1993.

55. Hindley, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146, p. 29-60, December 1969.

56. Hoare, C.A.R. The Emperor's Old Clothes, 1980 Turing Award Lecture. In *ACM Turing Award Lectures: The First Twenty Years 1966-1985,* p. 143-161, ACM Press Anthology Series, ACM Press/Addison-Wesley, New York, 1987.

57. Hunt, J.W. and T.G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences, *Communications of the ACM*, 20(5), p. 350-353, May 1977.

58. Hölzle, U. *Adaptive Optimization in Self: Reconciling High Performance with Exploratory Programming*. Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California, August 1994. Also published as Sun Microsystems Laboratories Technical Report, SMLI TR-95-35, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, March 1995.

59. Hölzle, U. and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI),* p. 326-336, Orlando, Florida, June 1994. Published as *SIGPLAN Notices 29(6)*.

60. Ingalls, D.H.H. A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 347-349, Portland, Oregon, September 1986.

61. Ingalls, D.H.H. *Object-Oriented Programming*. Video tape: Distinguished Lecture Series, Vol. II, Industry Leaders in Computer Science and Electrical Engineering, recorded July 19, 1989. University Video Communications, P.O. Box 5129, Stanford, CA 94309.

62. Johnson, R.E., Type-Checking Smalltalk. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 315-321, Portland, Oregon, September 1986.

63. Johnson, R.E., J.O. Graver, and L.W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA'88 Conference on Object-Oriented Programming Systems, Languages and Applications*, p. 18-26, San Diego, California, September 1988.

64. Johnson, S.C. Lint, a C Program Checker, July 26, 1978. In *UNIX Time-Sharing System: UNIX Programmer's Manual*, p. 278-290, Seventh Edition, Volume 2, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. Published by Holt, Rinehart and Winston, 1979.

65. Jones, N.D. and S.S. Muchnick. Binding Time Optimization in Programming Languages: Some Thoughts Towards the Design of an Ideal Language. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, p. 77-94, Atlanta, Georgia, January 1976.

66. Kanellakis, P.C., H.G. Mairson, and J.C. Mitchell. Unification and ML Type Reconstruction. In J.-L. Lassez and G.D. Plotkin (Eds.) *Computational Logic: Essays in Honor of Alan Robinson*, p.444-478, MIT Press, Cambridge, Massachussets, 1991.

67. Kaplan, M.A. and J.D. Ullman. A General Scheme for the Automatic Inference of Variable Types. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, p. 60-75, Tucson, Arizona, January 1978.

68. Kaser, O., C.R. Ramakrishnan, and S. Pawagi. On the Conversion of Indirect Recursion to Direct Recursion. *ACM Letters on Programming Languages and Systems*, 2(1-4), p. 151-164, March-December 1993.

69. Kernighan, B.W. and D.M. Ritchie. *The C Programming Language.* 2nd ed, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

70. Khedker, U.P. and D.M. Dhamdhere. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 16(5), p. 1472-1511, September 1994.

71. Krasner, G. (ed.), *Smalltalk-80: Bits of History and Words of Advice.* Addison-Wesley, Reading, Massachusetts, 1983.

72. Kumar, S., D.P. Agrawal, and S.P. Iyer. An Improved Type-Inference Algorithm to Expose Parallelism in Object-Oriented Programs. In B.K. Szymanski and B. Sinharoy (Eds.) *Languages, Compilers and Run-Time Systems for Scalable Computers*, p. 283-286, Kluwer Academic Publishers, Boston, Massachusetts, May 1995.

73. Lieberherrr, K.J. and I. Holland. Formulations and Benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3), p. 67-78, March 1989.

74. Madsen, O.L. *Abstract Syntax, Structure Editing and Type Browsing for Self*. In preparation, July 1995.

75. Madsen, O.L. and B. Møller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA'89 Object-Oriented Programming Systems, Languages and Applications*, p. 397-406, New Orleans, Louisiana, October 1989.

76. Madsen, O.L., B. Magnusson, and B. Møller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. In *OOPSLA/ECOOP'90 Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, p. 140-150, Ottawa, Canada, October 1990.

77. Madsen, O.L., B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Language.* ACM Press/Addison-Wesley, Workingham, England, 1993.

78. McConnell, C. and R. Johnson. Using Static Single Assignment Form in a Code Optimizer. *ACM Letters on Programming Languages and Systems*, 1(2), p. 152-160, June 1992.

79. Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, New York, 1988.

80. Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, p. 348-375, 1978.

81. Mitchell, J.C. *Foundations for Programming Languages*. To be published by MIT Press in 1996.

82. Moore, I., M. Wolczko, and T. Hopkins. Babel—A Translator from Smalltalk into CLOS. In *Technology of Object-Oriented Languages and Systems TOOLS-14*, p. 425-433, Santa Barbara, California, August 1994.

83. Muchnick, S.S. and N.D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

84. Oxhøj, N. *Practical Object-Oriented Type Inference.* M.S. thesis, Computer Science Department, Aarhus University, Ny Munkegade, DK-8200 Århus, Denmark, 1992.

85. Oxhøj, N., J. Palsberg, and M.I. Schwartzbach. Making Type Inference Practical. In *ECOOP'92, Sixth European Conference on Object-Oriented Programming*, p. 329-349, Utrecht, The Netherlands, June 1992. Springer-Verlag (LNCS 615).

86. Palsberg, J. and M.I. Schwartzbach, Object-Oriented Type Inference. In *OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, p. 146-161, Phoenix, Arizona, October 1991.

87. Palsberg, J. and M.I. Schwartzbach. *Polyvariant Analysis of the Untyped Lambda Calculus.* Technical Report, Daimi PB-386, Computer Science Department, Aarhus University, Århus, Denmark, 1992.

88. Palsberg, J. and M.I. Schwartzbach, *Object-Oriented Type Systems.* John Wiley & Sons, Chichester, England, 1993.

89. Pande, H.M. and B.G. Ryder. *Static Type Determination and Aliasing for C++*. Technical Report LCSR-TR-236, Rutgers University, December 1994.

90. ParcPlace Systems. *ObjectWorks Smalltalk User's Guide*. Release 4.1, Section 16: Deploying an Application, 1992.

91. ParcPlace Systems. *ObjectWorks Smalltalk User's Guide*. Release 4.1, Section 27: Binary Object Streaming Service, 1992.

92. Phillips, G. and T. Shepard. *Static Typing Without Explicit Types*. Unpublished report, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, 1994.

93. Plevyak, J. and A.A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications*, p. 324-340, Portland, Oregon, October 1994.

94. Plevyak, J. and A.A. Chien. *Iterative Flow Analysis.* Unpublished report, July 1995.

95. Plevyak, J. and A.A. Chien. Type Directed Cloning for Object-Oriented Programs. In *Proceedings of the workshop for Languages and Compilers for Parallel Computers*, Columbus, Ohio, August 1995.

96. Plevyak, J., X. Zhang, and A.A. Chien. Obtaining Sequential Efficiency in Concurrent Object-Oriented Programs. In *Conference Record of POPL'95: 22nd ACM Symposium on Principles of Programming Languages*, p. 311-321, San Francisco, California, January 1995.

97. Ponder, C. and B. Bush. Polymorphism Considered Harmful. *ACM SIGPLAN Notices*, 27(6), p. 76-79, June 1992.

98. Rémy, D. Typechecking Records and Variants in a Natural Extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, p. 77-87, Austin, Texas, January 1989.

99. Reps, T., S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM Symposium on Principles of Programming Languages*, p. 49-61, San Francisco, California, January 1995.

100. Reynolds, J.C. The Discoveries of Continuations. *Lisp and Symbolic Computation,* 6(3/4), p. 233-248, Kluwer Academic Publishers, November 1993.

101. Rivest, R.L., A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, 21(2), p. 120-126, February 1978.

102. Sakkinen, M. *A Comment on Unreachable Procedures.* Manuscript, May 1994.

103. Sanella, M., J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software - Practice and Experience* 23(5), p. 529-566, May 1993.

104. Schaffert, C., T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications*, p. 9-16, Portland, Oregon, September 1986.

105. Sharir, M., and A. Pnueli. Two Aproaches to Interprocedural Data Flow Analysis. In [83], p. 189-236.

106. Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, May 1991.

107. Shivers, O. Data-Flow Analysis and Type Recovery in Scheme. In P. Lee (Ed.) *Topics In Advanced Language Implementation*, MIT Press, Cambridge, Massachusetts, 1991.

108. Smith, R.B. and D. Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP'95, Ninth European Conference on Object-Oriented Programming,* p. 303-330, Århus, Denmark, August 1995. Springer-Verlag (LNCS 952).

109. Srivastava, A. Unreachable Procedures in Object-Oriented Programming. *ACM Letters on Programming Languages and Systems*, 1(4), p. 355-364, December 1992.

110. Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

111. Suzuki, N. Inferring Types in Smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages,* p. 187-199, Williamsburg, Virginia, January 1981.

112. Suzuki, N. and M. Terada. Creating Efficient Systems for Object-Oriented Languages. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages,* p. 290-296, Salt Lake City, Utah, January 1984.

113. Südholt, M. and C. Steigner. On Interprocedural Data Flow Analysis for Object Oriented Languages. In *Proceedings of CC'92, 4'th International Conference on Compiler Construction,* p. 156-162, Paderborn, Germany, October 1992. Springer-Verlag (LNCS 641).

114. Tenenbaum, A.M. *Type Determination for very High Level Languages*, Ph.D. thesis, Courant Institute of Mathematical Science, Computer Science Department, New York University, Report no. NSO-3, New York, October 1974.

115. Tiuryn, J. Type Inference Problems: A Survey. In *MFCS'90 Mathematical Foundations of Computer Science,* p. 105-120, Banská Bystrica, Czechoslovakia, August 1990.

116. Tofte, M. Type Inference for Polymorphic References. *Information and Computation*, 89(1), p. 1-34, November 1990.

117. Ungar, D. and R.B. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, June 1991. Also published as Sun Microsystems Laboratories Technical Report, SMLI TR-94-30, M/S 29-01, 2550 Garcia Avenue, Mountain View, CA 94043, USA, December 1994. Originally published in *OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, p. 227-241, Orlando, Florida, October 1987.

118. Ungar, D. Annotating Objects for Transport to Other Worlds. In *OOPSLA'95, Object-Oriented Programming Systems, Languages and Applications*, p. 73-87, Austin, Texas, October 1995.

119. Ungar, D. *How to Program Self 4.0*. Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043, USA, June 1995.

120. Vitek, J., N. Horspool, and J. S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proceedings of CC'92, 4'th International Conference on Compiler Construction*, p. 237-250, Paderborn, Germany, October 1992. Springer-Verlag (LNCS 641).

121. Wagner, T.A., V. Maverick, S.L. Graham, and M.A. Harrison. Accurate Static Estimators for Program Optimization. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, p. 85-96, Orlando, Florida, June 1994. Published as *SIGPLAN Notices 29(6)*.

122. Wand, M. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93(1), p. 1-15, July 1991.

123. Wegbreit, B. Property Extraction in Well-Founded Property Sets. *IEEE Transactions on Software Engineering*, SE-1(3), September 1978.

124. Wegman, M.N. and F.K. Zadeck. Constant Propagation With Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2), p. 181-210, April 1991.

125. Wegner, P. Dimensions of Object-Based Language Design. In *OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, p. 168-182, Orlando, Florida, October 1987.

126. White, J.L. Private conversation, February 1994.

127. Wilde, N. and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12), p. 1038-1044, December 1992.

128. Wilde, N., P. Matthews, and R. Huitt. Maintaining Object-Oriented Software. *IEEE Software*, 10(1), p. 75-80, January 1993.

129. Wilson, R.P. and M.S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, p. 1-12, La Jolla, California, June 1995. Published as *SIGPLAN Notices* 30(6), June 1995.

About the author

Ole Agesen received a Ph.D. in Computer Science from Stanford University in December 1995 for work[†] done in the Self project at Stanford University and Sun Microsystems Laboratories. Before joining the Self team, he received a Master's degree in Computer Science (Kandidatgrad) from Aarhus University, Denmark, in 1990[‡]. His research interests include object-oriented programming, languages, environments, and implementation techniques, including static program analysis.

[†] Among other important things, alpha-testing the Self Mug.

[‡] No, there was no Beta/Mjølner Mug, unfortunately.