The Cartesian Product Algorithm

Simple and Precise Type Inference Of Parametric Polymorphism

Ole Agesen
Computer Science Department, Stanford University
Stanford, CA 94305
agesen@cs.stanford.edu

Abstract. Concrete types and abstract types are different and serve different purposes. Concrete types, the focus of this paper, are essential to support compilation, application delivery, and debugging in object-oriented environments. Concrete types should not be obtained from explicit type declarations because their presence limits polymorphism unacceptably. This leaves us with type inference. Unfortunately, while polymorphism demands the use of type inference, it has also been the hardest challenge for type inference.

We review previous type inference algorithms that analyze code with parametric polymorphism and then present a new one: the cartesian product algorithm. It improves precision and efficiency over previous algorithms and deals directly with inheritance, rather than relying on a preprocessor to expand it away. Last, but not least, it is conceptually simple.

The cartesian product algorithm has been used in the Self system since late 1993. We present measurements to document its performance and compare it against several previous algorithms.

Keywords: concrete types, abstract types, type inference, polymorphism, inheritance, Self.

1 Introduction

1.1 Concrete versus Abstract Types

Concrete types (a.k.a. implementation types or representation types) are sets of classes. For example, an expression that returns instances of class `ListStack` has concrete type `{ListStack}`, and an expression that returns instances of either class `ArrayStack` or `ListStack` has concrete type `{ArrayStack, ListStack}`. Concrete types provide detailed information since they distinguish even different implementations of the same abstract data type.

In contrast, *abstract* types (a.k.a. interface types or principal types) capture abstract properties of objects. They may not distinguish between different implementations of the same abstract data type: both list-based and array-based stacks may have an abstract type like `[push:elmType→void; pop:void→elmType]`. Probably the best known inference algorithm for abstract types is Milner's [16].

Concrete and abstract types are extremes in a spectrum of type systems. Class-types, as found in BETA, C++, and Eiffel, are neither fully abstract (it is impossible to express the type of *any* object that has a push and a pop operation), nor fully concrete (declaring an object with class type `Stack`, does not reveal the specific subclass of which it is an instance).

Concrete types are the focus of this paper. Previous publications have argued that they directly support solutions to several important problems in object-oriented programming environments:

- *Compilation.* Concrete types support important optimizations such as elimination of dynamic dispatch and method inlining [17, 21].
- *Extraction.* Concrete types support delivery of compact applications by enabling an extractor to sift through an image of objects, selecting those that are essential to run a given application [4].
- *Browsing/debugging.* Concrete types help a programmer understand programs by allowing a browser to track control flow (and thereby data flow) through dynamically dispatched message sends [3].

The need for concrete type information may be greater in object-oriented environments than in "conventional" (C-like) environments: object-oriented programs consist of many small methods (hence, inlining is important); emphasis on code reuse yields large and general class libraries (hence, extraction is important); and understanding object-oriented programs is harder because control- and data-flow are coupled (hence, browsers should assist).

Even statically-typed languages like C++ can use concrete type information since the above three problems are not supported well by class types. For instance, knowing that an object is an instance of class `Stack`, is generally not sufficient to inline operations on it (the program may contain several subclasses of `Stack` and from the static type declaration we cannot tell which specific one occurs — but concrete type inference may tell) [13].

1.2 Polymorphism

Polymorphism, the ability of a piece of code to work on several kinds of objects, is a central part of object-oriented programming. Polymorphism is desirable because it enables more code reuse. We distinguish between two kinds. *Parametric polymorphism* is the ability of routines to be invoked on arguments of several types. A length function that works on both singly-linked and doubly-linked lists exhibits parametric polymorphism. *Data polymorphism* is the ability to store objects of different types in a variable or slot[1]. "Link" objects forming a heterogeneous list of integers, floats, and strings, exhibit data polymorphism because their "contents" slots contain objects of several types.

Explicit (programmer-inserted) *concrete* type declarations are undesirable because they limit polymorphism and code reuse. For example, adding the type declaration {Int, Float} to the formal arguments of a `max` routine makes it less reusable, because it can no longer be used to find the larger of two strings. Given that explicit concrete type declarations are undesirable, the considerable interest in *inference* algorithms is not surprising [3, 11, 17, 18, 19, 20, 25]. Ironically, while the desire to maximize polymor-

1. Slots in Self provide the functionality of instance variables, parents, formal arguments, and local variables.

phism demands type inference, polymorphism is also the hardest challenge for inference algorithms. To illustrate this, we applied a simple inference algorithm ("the basic algorithm," see Section 2.2) to the Self expression `30 factorial`, where the `factorial` method for an integer receiver is defined by

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

Rather than {smallInt, bigInt}, this very polymorphic and imprecise type was reported:

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

In fairness, the situation is more complex than just analyzing `factorial`. All methods invoked by `factorial`, and methods they in turn invoke, etc., must be analyzed. Furthermore, arithmetic overflows cannot be ruled out, forcing analysis of the bigInt implementation which uses lists, byte vectors, etc. In total, approximately a thousand lines of code must be analyzed to find the type of `factorial`. While some of this complexity is peculiar to the Self system and better results may be obtained on other systems, the example clearly demonstrates the insufficient precision of the basic algorithm when analyzing polymorphic code.

To address the imprecision, more accurate algorithms were developed; see [1] for a survey. Some of these can infer the precise type {smallInt, bigInt} for the factorial example. The improved precision came at a cost, though. Type inference got less efficient and more complex to understand. For instance, Phillips and Shepard implemented one of the more precise algorithms and found that it took 10 hours to infer types for the expression `3+4` in ParcPlace Smalltalk [19].

1.3 Contributions

In this paper we propose a new algorithm for analyzing *parametric* polymorphism. The algorithm applies the cartesian product to break the analysis of each send into a case analysis (each case represents a monomorphic combination of actual arguments). This way it simultaneously achieves precision and efficiency, e.g., for `30 factorial` it infers the type {smallInt, bigInt} in 9 seconds; see Section 4. More specifically, the cartesian product algorithm offers these advantages:

- it is precise — in some cases more than all previously published algorithms.
- it is efficient since it simultaneously avoids redundant analysis and iteration, the major performance limitations of previous algorithms.
- it is general; instead of applying the traditional stopgap of expanding away inheritance, the new algorithm handles it directly, permitting precise analysis of any combination of single/multiple/static/dynamic inheritance, inheritance of state (often found in prototype-based languages), and multiple dispatch.
- it is conceptually simple since it involves no expansions or iteration.
- it has been validated on real programs and used in the Self system since late 1993.

We do not consider data polymorphism in this paper, except briefly in the conclusions. For succinctness, from now on we will refer to "parametric polymorphism" as simply "polymorphism" and "concrete type" as "type."

The rest of this paper is organized as follows. Section 2 presents background material, including an overview of previous algorithms. Section 3 presents in detail the new algorithm and compares it against previous algorithms. Section 4 compares the algorithms using measurements obtained on the Self system. Section 5 offers our conclusions and outlines possible directions for future work. Finally, an appendix briefly lists a number of issues that lack of space prevents us from discussing in detail.

2 Background

The algorithms we describe are language independent, but we use Self as a source for examples and measurements [2, 24]. Detailed knowledge of Self is not required, although basic familiarity with Self or Smalltalk syntax and programming style is helpful.

This section reviews background material. Section 2.1 defines concrete types in a prototype-based context. Section 2.2 describes the basic type inference algorithm as a flow analysis. Section 2.3 introduces templates, a high-level way to understand the algorithms. Templates are used both to review previous algorithms and present the new algorithm. Section 2.4 pinpoints why polymorphism is hard to analyze. Finally, Section 2.5 reviews how previous algorithms deal with polymorphism. A more detailed and complete version of this background material is found in [1].

2.1 Definition of Type

In the introduction we informally presented types as sets of classes, following Suzuki and Johnson [23, 15]. Since Self is prototype-based and has no classes we need to modify the definition of type slightly. A Self *program* is a finite set of objects ("prototypes") $\{\omega_1, \omega_2, ..., \omega_n\}$. For example, ω_7 may be `true` and ω_9 may be `point`. A *program execution* is the computation that results from sending a designated message, say `main`, to a particular object, the *main object*. The message invokes the *main method* which in turn may invoke other methods. The main method is equivalent to the function `main()` in a C program.

Conceptually, the prototypes are created before program execution starts. All other objects participating in an execution are created by cloning a prototype or another object which has been recursively cloned from a prototype. The transitive closure of the relation "cloned-from" partitions objects into n *clone families*, one per prototype. We write $\overline{\omega}_i$ for ω_i's clone family. A *concrete type* T is an arbitrary subset of $U = \{\overline{\omega}_1, \overline{\omega}_2, ..., \overline{\omega}_n\}$, i.e., it is a set of clone families. We take the standard interpretation that an expression (slot) *has* type T if it only evaluates to (holds) objects belonging to one of the clone families in T. Examples of types are:

- type(`x<y`) = {$\overline{\texttt{true}}$, $\overline{\texttt{false}}$}
- type(`30 factorial`) = {$\overline{\texttt{smallInt}}$, $\overline{\texttt{bigInt}}$}. Without range analysis or constant folding we cannot say that whether a small integer or a big integer may result.

- type(nil) = {$\overline{\texttt{nil}}$}. "The type of the expression nil, which returns the nil object, is {$\overline{\texttt{nil}}$}." Even if a language, like C++, does not treat nil (actually NULL) as an object, it may be worth doing so during type inference. The prize is a complimentary "NULL-analysis": types that include $\overline{\texttt{NULL}}$ identify where NULL pointers may show up during execution.

By definition, if an expression or a slot has type T, it also has type T′ for any T′⊇T. This 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! Indeed, the goal is to infer the most precise types where a more precise type is a smaller set.

In a program with n prototypes, there are 2^n possible concrete types. In contrast, a C++ programmer writing "semi-abstract" type declarations is only granted n possible types (one per class). The precision difference between an exponential and a linear number of types is significant. There are concrete types that the C++ programmer cannot express, yet our type inference algorithm can infer.

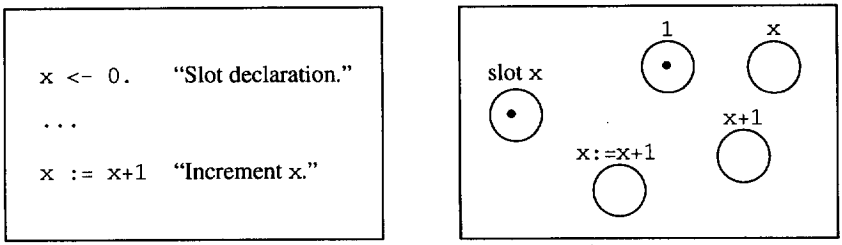
2.2 The Basic Type Inference Algorithm

The basic type inference algorithm was presented by Palsberg and Schwartzbach [18] as a constraint-solving problem: derive a set of constraints from the program being analyzed, solve the constraints using a fix-point algorithm, and finally map the solution back onto the program to obtain the desired type information. The algorithm has deficiencies when analyzing polymorphic code, but constitutes the core of the improved algorithms, so we review it here. 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 [9] perspective allows a direct correspondence between analyzing a program and executing it, making the analysis algorithms easier to understand. Referring to the program being analyzed as the *target program*, there are three steps in the basic algorithm.

Step 1. Allocate type variables. The first step is to associate a *type variable* with every slot and expression in the target program (see Fig. 1). A type variable is simply a variable whose possible values are types (i.e., sets of clone families). Initially all type variables are empty, but the next two steps adds enough clone families to them to make them hold sound types for the slots and expressions they are associated with. Nothing is ever removed from a type variable.

Step 2. Seed type variables. The second step adds a single clone family to certain type variables: the type variables are *seeded*. The goal is to capture the *initial state* of the target program by initializing type variables that correspond to slots or expressions where objects are found initially. For example, the type variable for the slot x←nil becomes {$\overline{\texttt{nil}}$}. Similarly, the type variable for a literal object such as 'bicycle' has $\overline{\texttt{string}}$ added to it. In Fig. 1, a seeded type variable's member is shown as a small dot.

Step 3. Establish constraints and propagate. The last 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 a run-time data flow.



Target program. Type variables.

Fig. 1. A type variable is associated with every slot and expression in the target program. A program fragment is shown in the left frame, and the type variables (in no particular order) in the right frame. The type variable for the *slot* x is labelled "slot x" to distinguish it from the *expression* x which reads this slot.

For example, if the target program executes the assignment `x:=exp` there is a data flow from `exp` to `x`. Intuitively, the data flow means that any object that may result from evaluating `exp` can also be in the `x` slot. When the algorithm encounters this data flow, an edge is added from `exp`'s type variable to `x`'s type variable, reflecting that type(`exp`) $\subseteq$ type(`x`). Fig. 2 depicts this.

Fig. 2. The situation before and after establishing the constraint for `x:=exp` and propagating.

Whenever an edge is added to the graph, clone families are *propagated* along it. In Fig. 2 all clone families in the `exp` node are pushed along the arrow to the `x` node. As more and more constraints are added to the network, the clone families that were originally only in the seeded type variables can flow further and further. The eager propagation ensures that subset relations such as type(`exp`) $\subseteq$ type(`x`) always hold: if a clone family is added to type(`exp`), it is immediately propagated to type(`x`), reestablishing the subset relation.

Adding constraints makes more propagation necessary. 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 are needed. Thus, Step 3 consists in repeatedly establishing constraints and propagating, until no more can be done.

To ensure soundness, a constraint must be generated for *every* possible data flow in the target program [1]. Different languages have different constructs that produce data flows, but some general examples can be given (more examples are found in [1, 3, 17, 18, 20]):

- Assignments generate data flows from the new value expressions to the assigned variables (Fig. 2).
- Variable reads generate data flows from the accessed variables to the accessing expressions.

- Message sends generate data flows from the actual argument expressions (including the receiver) to the formal arguments of the invoked methods. Furthermore, data flows return the results of the invoked methods to the message sends.
- Primitive data types like integers and floats, and their operations, including built-in control structures.

2.3 Templates

In [1] we introduced templates and used them to study how several previously published type inference algorithms analyze polymorphism. Templates also enable a concise exposition of our new algorithm, so we review them here. Just as methods raise the abstraction level by encapsulating statements, templates raise the abstraction level by encapsulating constraints. Starting with the unstructured constraint graph produced by the basic type inference algorithm, we can carve it 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, and formal arguments of M, and
- the edges (constraints) originating from these nodes.

Nodes corresponding 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). In the remainder of this paper we work at the template level, to avoid the complexity of dealing with a myriad of individual constraint. For an example of a template, consider the following `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 briefly in the appendix):

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

Fig. 3 shows the template for this method. It is a box with two input type variables at the top, corresponding to the formal arguments `self` and `a`, and a type variable for the result at the bottom. The result type is determined by contributions from two `ifTrue:False:`-templates (a conditional statement in Self is a send of `ifTrue:False:` to `false` or `true`, hence there are connections to templates for the `ifTrue:False:` methods in `false` and `true`, respectively). To simplify the picture, we omitted the constraints for the comparison `self>a` and the constraints propagating its result into the `self` slots of the `ifTrue:False:`-templates. Fig. 3 illustrates how a send, `3 max: 4`, is connected to the `max:`-template. The inference algorithm determines the send's type by propagating the actual argument types through the template(s) of the method(s) that may be invoked. When a send may invoke several methods, its type is the union of the result types of these methods.

2.4 Polymorphism is Analyzed Imprecisely by the Basic Algorithm

We now turn our attention to polymorphic methods. Fig. 4 shows what happens when the target program contains two sends of `max:`, one with integer arguments and the other with float arguments. The basic algorithm infers the type $\{\overline{\texttt{smallInt}}, \overline{\texttt{float}}\}$ for both sends. This is imprecise since invoking `max:` on two integers can "obviously"

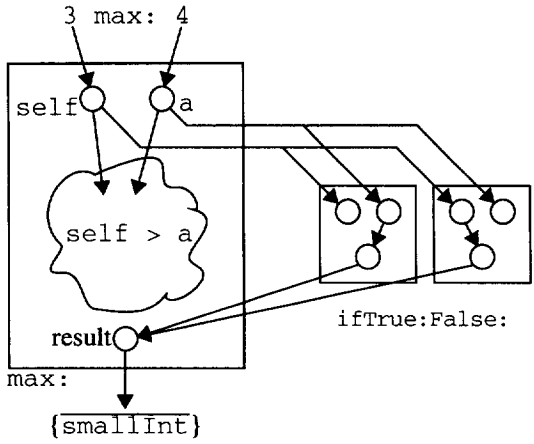


Fig. 3. A template for the max: method and a send connecting to it. The type of the send is determined by propagating the types of actual arguments through the template.

not produce a float or vice versa. The problem is that different types, {smallInt} and {float}, merge to produce the type {smallInt, float} in the input type variables of the max:-template. The merge causes information loss: once merged, the types cannot (easily) be separated and the result is imprecise types for all sends invoking max: with less than the full union type.

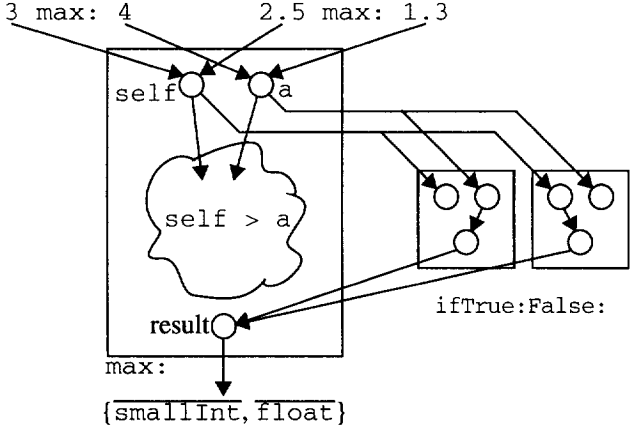


Fig. 4. The basic algorithm infers the imprecise type {smallInt, float} for the above sends of max:, even though one computes the max of two integers and the other the max of two floats.

2.5 Previous Improvements of the Basic Algorithm

The problem with the basic algorithm is that it creates only one max:-template. Its type must then be general (or imprecise) enough to accommodate all uses of max:. An algorithm, like the basic one, that analyzes each method once is called *monovariant*. The improved algorithms are all *polyvariant*: they may analyze methods multiple times by creating more than one template for them.

2.5.1 1-Level and p-Level Expansion Algorithms

Palsberg and Schwartzbach saw the limitations of the basic algorithm and proposed the following polyvariant improvement [17]. *1-level expansion:* always connect different sends in the target program to different templates. For example, if there are two `max:`-sends in the target program, two `max:`-templates are created, one for each send. The intuition is that since two different sends may supply actual arguments of different types, they should not share a template because this causes the types to merge.

The intuition is good, but there are two problems with the 1-level expansion. First, it is inefficient due to redundant analysis. Different sends with the same selector often supply the same argument types, e.g., most sends of + add integers. The 1-level expansion algorithm does not recognize this, and unnecessarily re-analyzes the integer addition method for each such send. (The Self integer addition method is not trivial since it handles type coercions and overflows into bigInts). Second, precision is still inadequate. Fig. 5 shows that the 1-level expansion is just as imprecise as the basic algorithm when applied to the two sends computing maxima of integers and floats. The only difference is that the merging now happens in a pair of `ifTrue:False:`-templates that are shared between two `max:`-templates. These templates are shared because there is only one send of `ifTrue:False:` in the `max:` method.

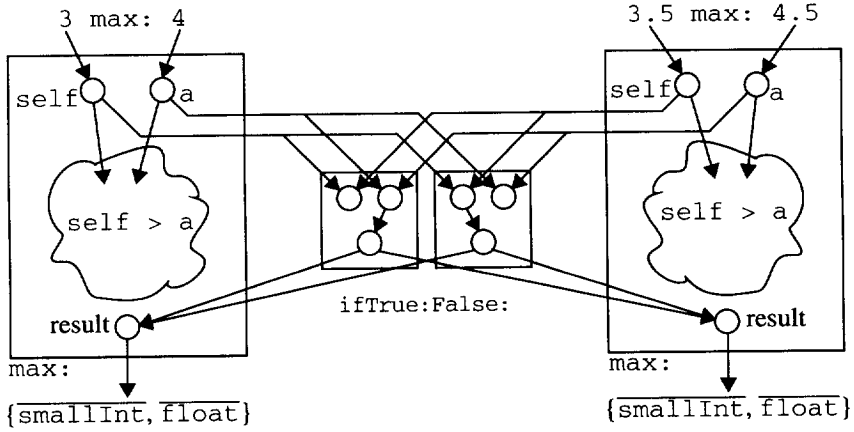


Fig. 5. The 1-level expansion algorithm creates a `max:`-template for each of the two sends, but there is only a single send of `ifTrue:False:` so the two `max:`-templates must share a pair of `ifTrue:False:`-templates and the types still merge.

The obvious way to improve precision is to avoid sharing the `ifTrue:False:`-templates, e.g., by using a *2-level expansion.* While this allows precise type inference for the situation shown in Fig. 5, it is even more inefficient since the second expansion further worsens the redundancy problems. Moreover, many code fragments still cannot be precisely analyzed, e.g., the following method that invokes `max:` twice to compute the max of three numbers, the receiver, `x` and `y`:

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

This method defeats the 2-level expansion by having a 3-deep polymorphic call chain: `max:Max:`→`max:`→`ifTrue:False:`. In general, a *p-level expansion* algorithm is precise if polymorphic call chains are at most p calls deep, but the worst-case com-

plexity is exponential in p since each expansion incurs a quadratic cost increase [17]. Vitek et al. [25] described this algorithm using the concept of call strings, and Phillips and Shepard [19] applied it to ParcPlace Smalltalk.

2.5.2 The Iterative Algorithm

With the iterative algorithm, Plevyak and Chien [20] broke the unfavorable precision/efficiency trade-off inherent in the expansion algorithms. They simultaneously improved precision and efficiency by allowing a variable amount of expansion in "different parts" of the program. To understand their algorithm, assume we are analyzing these two sends

```
rcvr1 max: arg1.        rcvr2 max: arg2.
```

Consider now this question: *"when is it safe to connect the two sends to a shared template?"* The answer is that the two sends can share a template safely, i.e., without loss of precision, if:

$$\text{type}(\texttt{rcvr}_1) = \text{type}(\texttt{rcvr}_2) \text{ and } \text{type}(\texttt{arg}_1) = \text{type}(\texttt{arg}_2).$$

We call an algorithm which answers the sharing question based upon the above test "ideal." It is precise because it never allows different types to merge, and 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. In other words, the ideal algorithm needs to know the types before it can compute them!

Plevyak and Chien found a way out of this dilemma by iterating the type inference. The first iteration is the basic algorithm. In subsequent iterations, the ideal algorithm's sharing test is approximated using the types of the previous iteration. That is, the two sends can share a template in iteration p, if:

$$\text{type}_{p-1}(\texttt{rcvr}_1) = \text{type}_{p-1}(\texttt{rcvr}_2) \text{ and } \text{type}_{p-1}(\texttt{arg}_1) = \text{type}_{p-1}(\texttt{arg}_2).$$

The iterative algorithm is precise since it can handle arbitrarily deep polymorphic call chains, given enough iterations: after p iterations, the precision is the same as that of the p-level expansion algorithm. The iterative algorithm achieves this precision at a lower expected computational cost because it "selectively" expands the program as is needed to avoid non-equal types merging at calls.

Iteration, while improving precision, also has drawbacks. One is the overhead of iteration, possibly slowing down type inference by a factor roughly equal to the number of iterations required. Implementation complexity may also be an issue: it is non-trivial to map type information across iterations, because the program is expanded differently. In particular, maintaining validity of the previous iteration's type information downstream from a method that is further expanded in the current iteration is a challenge. Finally, it can be hard to know when to stop iterating. In principle, once $p \geq r$, where r is the length of the longest polymorphic call chain, no further precision improvements are possible. In reality, the situation is more complex, due to block specialization and recursion [20] (see also the appendix).

3 The Cartesian Product Algorithm

The previous algorithms aim 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 harvest timely the type information needed during analysis. The cartesian product algorithm (CPA for short) is fundamentally different. It does not partition sends, but instead turns the analysis of *each* send into a case analysis. Given a send to analyze, 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 makes exact type information available for each case immediately, thus eliminates 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 mainly compare it 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 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. 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 call, only polymorphic call sites.

CPA, unlike the previous algorithms, exploits this observation. CPA creates *monomorphic templates* only. Given a send such as

```
rcvr max: arg.
```

Let R = type(`rcvr`) and A = type(`arg`). Suppose it is (somehow) known that

$R = \{r_1, r_2, ..., r_s\}$ and
$A = \{a_1, a_2, ..., a_t\}$.

In general, of course, these types will not be fully known during type inference; this was the motivation behind the iterative algorithm. In Section 3.1 we explain how it is nevertheless possible to use the types without iterating or even approximating.

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 arguments.

$$R \times A = \{(r_1,a_1), ..., (r_1,a_t), ..., (r_i,a_j),..., (r_s,a_1), ..., (r_s,a_t)\}.$$

Next, CPA propagates each $(r_i,a_j) \in R \times A$ into a separate `max:`-template. If a `max:`-template already exists for a given (r_i,a_j) pair, it is reused; if no such template exists, a new one is created and made available for this and future (r_i,a_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. Fig. 6 illustrates the situation with concrete objects for a single send.

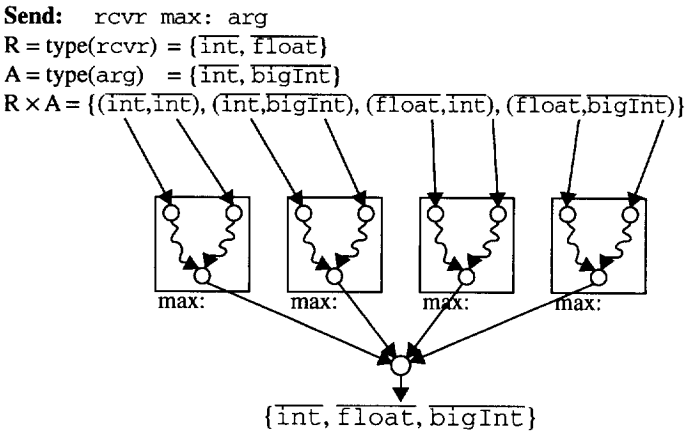


Fig. 6. CPA forms the cartesian product of the receiver type and argument types of the send it is analyzing. Each member of the product is propagated to a template reserved exclusively for that receiver and argument combination. The result type of the send is obtained by collecting the contributions from each template.

In the Self implementation we use a "dictionary" per method to organize the templates. All templates for a method M, no matter which send invoking M caused their creation, are stored in M's dictionary. The dictionaries are keyed 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 methods M_i (i = 0, 1, ..., q) that the send may invoke.
2. Generate the cartesian product (lazily, as described in Section 3.1).
3. For each method M_i and each tuple $(a_0, ..., a_k)$ in the cartesian product, look up $(a_0, ..., a_k)$ in M's template dictionary. If no matching template is found, create one and add it to the dictionary.
4. Connect the send to the — new or old — template.

To obtain an efficient (and terminating!) algorithm, it is necessary to maintain per-method pools of templates rather than per-send pools, to ensure that different sends whose cartesian products have tuples in common can share templates. It is worth noting that the template dictionaries 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. This only becomes clear gradually, as the analysis is progressing.

3.1 How CPA Can Use Exact Type Information Without Iterating

We have yet to explain how CPA 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.

How is this achieved? The key is that, while the iterative algorithm is comparing 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 only partially known 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 goes back and extends the cartesian product with the new combinations that are now possible. This yields a correct solution since types grow monotonically during type inference.

For illustration, assume that we started with the situation shown in Fig. 6, but a new clone family, `rational`, has just arrived in the type of the argument. We first extend the cartesian product with two new pairs: (`int`, `rational`) and (`float`, `rational`). Then we propagate these pairs into (new or old) `max:`-templates and collect the new possible result types. Fig. 7 illustrates all this.

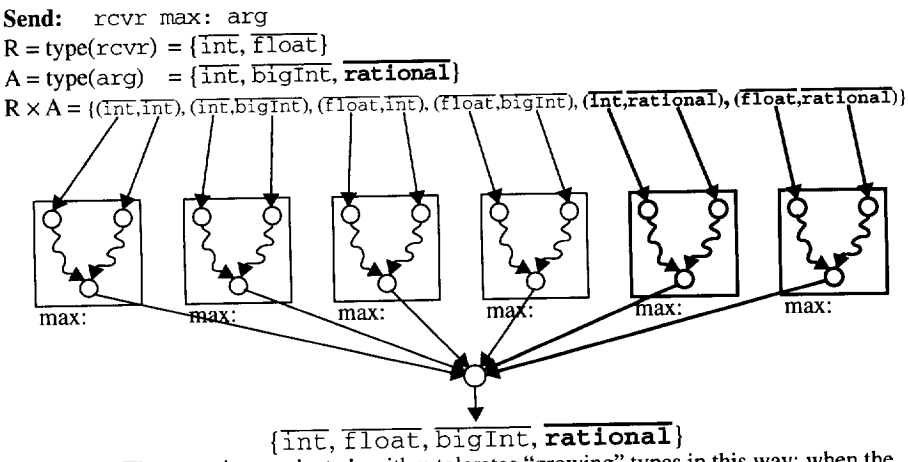


Fig. 7. The cartesian product algorithm tolerates "growing" types in this way: when the type of a receiver or an argument grows, the cartesian product is extended, and the send is connected to additional templates. The new structures are shown in bold above.

Two key properties allow CPA to avoid iteration. First, the cartesian product is a *monotonic* function. If either R or A grows, the cartesian product R × 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, Plevyak and Chien pays the price of iteration.

Seen in this light, the p-level expansion algorithms form a family of algorithms that apply particularly simple monotonic functions to decide whether or not to share. These functions do not depend on the types being inferred, but merely on the static program text. In [1] such algorithms are called *non-adaptive* because they uniformly apply the same effort throughout the target program rather than focusing on the parts where the most complex polymorphism is found. The iterative algorithm and CPA are *adaptive* because they use type information during inference to concentrate the analysis effort on the demanding parts of the target program.

The hash function algorithm, described in [1, 3], 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 for the non-receiver arguments. The hash algorithm is adaptive because it uses receiver types during analysis. Like the two algorithms of which it is a hybrid, it is monotone and can be implemented in one pass.

3.2 Assessing the Cartesian Product Algorithm

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

CPA is *efficient*, because redundant analysis is avoided. As we have seen before, the key to avoiding redundancy is sharing. Each time an (r_i,a_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. In contrast, the iterative algorithm may generate templates with overlapping types such as:

Template 1: type(`self`) = {`int`, `float`}
type(`arg`) = {`int`, `bigInt`, `rational`}

Template 2: type(`self`) = {`int`, `float`}
type(`arg`) = {`int`, `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, hence can be processed faster than templates with larger types.

3.3 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 with one argument (it is straightforward to generalize to an arbitrary number of arguments). Assume that the set of possible receivers and arguments for the method are drawn from these domains:

$P = \{r_1, r_2, ..., r_s\}$ (possible receivers)
$Q = \{a_1, a_2, ..., a_t\}$ (possible arguments)

In the worst case, the iterative algorithm may generate an exponential number of templates, namely 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, generates at most a polynomial number of templates, namely one for each member P×Q, i.e., s*t 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 {`smallInt`, `bigInt`, `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 3-5, 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 argument or receiver that fails the test is *megamorphic*. When megamorphic arguments occur, the full cartesian product should no longer be computed. Instead, we create tuples of the form (x_0, x_1, *, x_3) where a star indicates a megamorphic argument. When such a tuple is propagated into a template, the star is expanded to the full megamorphic type.

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 loss of precision. To limit the loss, templates should not be shared between sends with and without megamorphic arguments to avoid "polluting" the non-megamorphic cases. The loss can be further limited, e.g., by applying expansions or even iteration to the (few) megamorphic cases.

In practice the exact choice does not seem to matter much. We offer two possible explanations. First, megamorphism is rare. Measuring 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, he is probably exploiting some uniform behavior which in turn limits the loss of precision when the cases are merged during analysis.

3.4 A Case Where the Cartesian Product Algorithm Improves Precision

Even though both the iterative and the cartesian product algorithm avoid losing precision due to merging of types, the property that CPA analyzes each argument combination monomorphically 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

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

The iterative algorithm will infer that the type of `x mod: y` is {$\overline{\texttt{smallInt}}$, $\overline{\texttt{float}}$} whereas the cartesian product algorithm will infer the more precise type {$\overline{\texttt{smallInt}}$}. 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 {$\overline{\texttt{smallInt}}$, $\overline{\texttt{float}}$}. Then it proceeds to find that the type of `self div: arg` is {$\overline{\texttt{smallInt}}$} (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 {`smallInt`, `float`}, the type of the product is {`smallInt`, `float`}. In turn, the type inferred for `self-(arg*(self div: arg))` is {`smallInt`, `float`}, and finally it is found that the type of `x mod: y` is {`smallInt`, `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 {`smallInt`} straightforwardly has result type {`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 {`smallInt`} and three empty result types, to find that the type of `x mod: y` is {`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 [10]. 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.

3.5 Inheritance

To use terminology that most readers are familiar with, we give examples from a class-based language in this section. Inheritance is an important code reuse mechanism in object-oriented programming. Often, inheritance is used in a way that attains substitutability, i.e., ensures that an instance of a subclass can be used wherever an instance of the superclass is expected. Substitutability may also be enforced by the language, although it is not in Self. Even if inheritance is used in this restricted manner, the fact remains that expressions in inherited methods may have different types in the context of a subclass than in the context of a superclass. For instance, a message sent to self may be bound differently when the method is inherited, causing the type to change. Another, and simpler, example is a method that returns self, e.g., a `display` method implemented for points and inherited by coloured points. The left hand side of Fig. 8 illustrates this situation. Naively, the type inferred for `display` is {`Point`, `ColourPoint`}, because it may be applied to either a point or a coloured point. Propagating this type to a send that invokes `display` only on coloured points is imprecise: such a send should have the type {`ColourPoint`}. Compiler writers facing this issue found solutions such as customization [8] where the idea is to recompile a method for each class that inherits it to make the class of `self` known in the compiled code.

Previous type inference algorithms for object-oriented languages have achieved a similar effect by using a preprocessor to eliminate inheritance [10, 11, 17, 18]. The preprocessor copies methods down into the classes inheriting them. The result of this "expansion" is shown in the right hand side of Fig. 8: there are now two identical methods $\texttt{display}_{\texttt{Point}}$ and $\texttt{display}_{\texttt{ColourPoint}}$. The former applies to points only, thus has result type {`Point`}. The latter applies to coloured points only, and has result type {`ColourPoint`}. Now, if a send invokes `display` exclusively on coloured

points, its type is {ColourPoint}, since the send is connected to templates for $\texttt{display}_{\texttt{ColourPoint}}$ only.

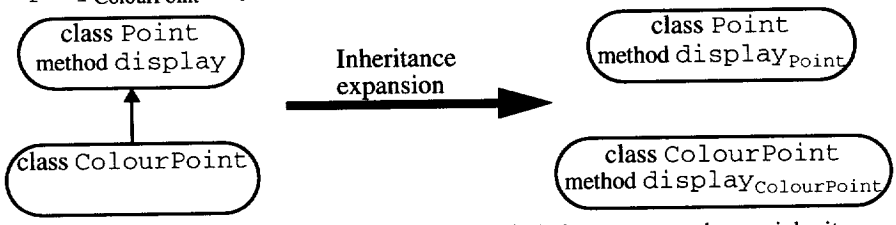


Fig. 8. Most type inference algorithms, but not CPA, have to expand away inheritance to gain precision. The expansion copies inherited methods down into subclasses.

Expanding away inheritance improves precision, because inaccuracies resulting from analyzing a single method in the context of several classes are avoided. The expansion can be expensive, though. In the worst case program size increases quadratically, even for programs using only single inheritance [18]. Care must also be taken to handle overridden methods correctly, including sends to `super` ("resends"). For Self, there are additional complications: it is non-trivial to eliminate inheritance of state, not to mention dynamic inheritance, by an expansion.

The cartesian product algorithm removes the need for the expansion. It can analyze an unexpanded program with the same precision as an expanded one. The reason is that CPA, unlike other algorithms, 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[2]. Avoiding the inheritance expansion has several advantages:

- *Simplicity.* We reuse an existing mechanism, the cartesian product, instead of applying a special-purpose mechanism to deal with inheritance.
- *Efficiency.* Templates are created only when needed, whereas the static expansion assumes that a method can be invoked on any class inheriting it. (Optimizing the expansion is possible, of course).
- *Generality.* The cartesian product handles inheritance of state, dynamic inheritance, and multiple inheritance. It even supports languages with multiple dispatch (multi-methods) [5, 6], by a straightforward extension to the single-dispatch case: when the cartesian product is generated, each tuple is propagated to a template for the *specific* method that implements the operation for the combination of arguments in the tuple.

3.6 Summary

Table 1 condenses the most important attributes of the algorithms that we have discussed. The hash algorithm [1, 3] is included to show where it fits in, even though we have only touched upon it briefly in this paper. The efficiency column is informal, since it is not based on extensive measurements or theoretical complexity studies of

2. The hash function algorithm also has this property [3], but is restricted to singly-dispatched languages.

the algorithms (the latter would require a statistical model of how "typical" programs use polymorphism).

Algorithm	Efficiency	Precision	Adv./contributions	Drawbacks
Basic	Fast	Inadequate	Simple, first	Monovariant (imprecise)
1-level expansion	Slow	Meager	First polyvariant algorithm	Fails on polymorphic call chains of length >1
p-level expansion	Too slow	Good	Parameterized cost/precision trade-off	Gets slow before reaching high precision
Hash	Fast	Good	Good handling of polymorphic receivers	Polymorphic arg's need ad-hoc rules
Iterative	Medium	Better	Better cost/precision trade-off	Iteration overhead, complex
CPA	Fast	Better still	Non-iterative and first to not partition sends	Scaling?

Table 1. Summary of six type inference algorithms.

4 Measurements

We have implemented and tested six type inference algorithms: the basic, the p-level expansion (p = 1, 2, 3, 4), and the cartesian product algorithm, but not the iterative algorithm[3]. The implementation consists of 10,000 lines of Self source code that, in addition to type inference, parses byte codes into expression trees, performs grouping (see appendix), incremental analysis, SSA transformation, and application extraction. We applied the six inference algorithms to four existing programs in the Self system:

- *Factorial:* invocation of the recursive factorial function shown in the introduction.
- *Richards:* an operating system simulator [7].
- *Deltablue:* a multi-way constraint solver [22].
- *Diff:* Mario Wolczko's Self version of the Unix diff command [12].

The programs were analyzed unmodified, with one exception: we had to patch the `_Mirror` primitive to return nothing instead of 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. The patch was removed when running the other inference algorithms since they are precise enough to avoid falling into the reflective trap[4].

3. We have not implemented the iterative algorithm for Self, hence cannot include it in the following direct comparison. We have refrained from comparing with the numbers published by Plevyak [20] because his system is implemented in a different language than Self (possibly less efficient), and he analyzes different benchmarks, written in a different language.

4. The pure form of the iterative algorithm would also need the patch, since the first iteration is the basic algorithm.

Like most programs, the above four 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 a bigInt implementation which comprises hundreds of lines. To obtain a more useful measure of program size, we applied the extractor described in [4], and measured the size of the resulting self-contained source files. When counting everything, including variable definitions, blank lines, names spaces, etc., Factorial is 1300 lines, Richards 1700 lines, Deltablue 1800 lines, and Diff 5000 lines. Counting method bodies only (i.e., statements and expressions), Factorial is 900 lines, Richards 1100 lines, Deltablue 1200 lines, and Diff 2300 lines.

The extractor is based on type inference 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. The results are shown in Table 2. In general, CPA delivers the smallest extractions, indicating that it is the most precise inference method. On Diff the expansion algorithm improves dramatically when p is increased from 1 to 2, and shows little additional gain past 2. The *increasing* line counts from p=1 to p=2 on Factorial, Richards, and Deltablue are explained by the removal of the reflective patch. The numbers in Table 2 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 [14]).

	Basic	1-level	2-level	3-level	4-level	CPA
Factorial	1692 / 678	1667 / 649	2847 / 608	2847 / 608	2847 / 608	1306 / 464
Richards	2117 / 786	2086 / 753	3259 / 698	3259 / 698	3259 / 698	1731 / 560
Deltablue	2199 / 844	2174 / 815	3394 / 784	3392 / 782	3392 / 782	1777 / 600
Diff	74816 / 1582	71598 / 1510	5578 / 1196	5576 / 1194	5547 / 1194	5447 / 1123

Table 2. Number of source lines/methods extracted.

Table 3 compares CPU times of the type inference algorithms, measured on a 50 MHz SparcStation 10. To focus on the type inference algorithms per se, the time spent on grouping objects is omitted. CPA is consistently fastest, although in fairness, it is also the best tuned of the implementations. Remarkably, it is faster than even the basic algorithm analyzing the simpler patched programs. For the expansion algorithms, two opposing trends battle as p increases. On the one hand, precision improves, enabling the analysis to stay clear of "irrelevant" code that is not part of the application. This speeds up type inference. On the other hand, redundancy increases, acting to slow down type inference. Beyond p=2, the effect of the former trend wears off, and the cost of redundancy shows through. Finally note that the basic algorithm's times are lower bounds for the iterative algorithm, since the first iteration is the basic algorithm.

	Basic	1-level	2-level	3-level	4-level	CPA
Factorial	11	19	17	32	78	9
Richards	21	24	19	37	94	9
Deltablue	22	27	22	43	112	17
Diff	117	250	50	126	377	38

Table 3. Type inference time in seconds (CPU time on a 50 MHz SparcStation 10).

Polyvariance is essential to obtain precision, but too much hurts efficiency. Table 4 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. The second number allows a large method to weigh heavier in the average. The polyvariance degree of CPA is between that of the 2- and 3-level expansion algorithms, yet CPA is faster than these algorithms, because monomorphic templates can be processed faster.

	Basic	1-level	2-level	3-level	4-level	CPA
Factorial	1.0 / 1.0	3.9 / 3.3	6.6 / 4.9	12.1 / 8.7	21.1 / 15.7	6.4 / 5.0
Richards	1.0 / 1.0	4.0 / 3.2	6.5 / 4.7	11.8 / 8.1	20.6 / 14.5	6.2 / 4.7
Deltablue	1.0 / 1.0	4.1 / 3.4	6.6 / 4.8	11.8 / 8.3	20.3 / 14.7	10.8 / 8.3
Diff	1.0 / 1.0	4.8 / 4.0	7.6 / 5.6	14.5 / 10.0	27.3 / 19.1	11.5 / 9.3

Table 4. Number of times each method/expression was analyzed.

Paying the price for a certain amount of polyvariance, it should be used well. Table 5 shows average type sizes over all expressions. The averages are over the polyvariant domains, i.e., if an expression was analyzed twice, it counts twice in the average. CPA achieves the lowest sizes, even compared against the 3- and 4-level expansion algorithms which analyze each expression more times (see Table 4). The expansion algorithms' improvements from p=1 to p=2 are particularly clear in Table 5.

	Basic	1-level	2-level	3-level	4-level	CPA
Factorial	13.41	5.58	1.71	1.67	1.69	1.03
Richards	17.56	8.54	1.85	1.79	1.78	1.04
Deltablue	17.56	8.31	1.92	1.84	1.84	1.11
Diff	37.89	19.0	2.25	2.13	2.13	1.34

Table 5. Average size of types for all expressions.

To conclude the empirical results, it is amusing to see the types inferred by each algorithm for the expression `30 factorial`. As shown in the introduction, the basic algorithm inferred:

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

The 1-level expansion algorithm did slightly better:

{smallInt, bigInt, collector, memoryState, byteVector, mutableString, immutableString, float, link, list, primitiveFailedError, userError, time, false, true, nil}.

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

{smallInt, bigInt, float, primitiveFailedError}.

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

{smallInt, bigInt}.

5 Conclusions

We have presented the cartesian product algorithm. It is based on the key insight that the cartesian product can be used to monotonically and soundly decompose the analysis of a send into a number of monomorphic cases. This decomposition accurately mir-

rors the situation at program execution time where activation records are monomorphic. It has several advantages:

- *Precision* is sharpened since some methods can be analyzed more precisely in monomorphic contexts. Measurements additionally show that CPA achieves the smallest average size of the inferred types.
- *Efficiency* is improved since monotonicity of the cartesian product allows a 1-pass implementation, yet no redundant analysis is done. Even sends that have only partially overlapping types can share analysis cases. Measurements show that CPA is faster than even the basic algorithm.
- *Generality* is improved. Unlike other algorithms that expand away inheritance, CPA handles it directly, permitting precise analysis of wider range of languages, including ones with inheritance of state, dynamic inheritance, multiple inheritance, and multiple dispatch.
- *Simplicity* is preserved. The core of the algorithm is the familiar cartesian product. Expansions are not required, inheritance elimination is unnecessary, and iteration with mapping of type information across differently expanded versions of the target program is avoided.

CPA addresses parametric polymorphism only. It does nothing to improve the analysis of code with data polymorphism, thus is no better (but also no worse) in this regard than the basic algorithm. What then can be done about data polymorphism? A number of algorithms have been published that target data polymorphism. Analogous to analyzing parametric polymorphism by copying methods, these algorithms copy classes in an attempt to keep different uses of (data) polymorphic instance variables separate. For example, Oxhøj et al. proposed a 1-level class expansion [17], Phillips and Shepard generalized this to p levels [19], and Plevyak and Chien developed an iterative algorithm [20].

While CPA itself does not address data polymorphism, neither does it conflict with the existing algorithms for analyzing data polymorphism. Hence, it is practical to 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.

Acknowledgments. Thanks to Lars Bak, Craig Chambers, Bay-Wei Chang, Ole Lehrmann Madsen, John C. Mitchell, Ivan Moore, David Ungar, Jan Vitek and Mario Wolczko for discussions and comments on earlier drafts of this paper, and to the referees for feedback on the submitted version. The author has been generously supported by The Natural Science Faculty of Aarhus University, The Danish Research Academy, and Sun Microsystems Laboratories. The Self project has been supported by the NSF PYI Grant #CCR-8657631, Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments and DEC.

References

1. Agesen, O., Constraint-Based Type Inference and Parametric Polymorphism. In *SAS'94, First International Static Analysis Symposium,* p. 78-100, Namur, Belgium, Sept. 1994. Springer-Verlag, LNCS 864.
2. Agesen, O., L. Bak, C. Chambers, B.W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, M. Wolczko. *How to use Self 3.0 & The Self 3.0 Programmer's Reference Manual.* 1993. Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043, USA. Available by anon. ftp from self.stanford.edu or www: http://self.stanford.edu/.
3. 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, July 1993. Springer-Verlag, LNCS 707.
4. Agesen, O. and D. Ungar, Sifting Out the Gold: Delivering Compact Applications from an Exploratory Object-Oriented Programming Environment. In *OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications,* p. 355-370, Portland, Oregon, Oct. 1994.
5. Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 17-29, Portland, Oregon, Sept. 1986.
6. Chambers, C., Object-Oriented Multi-Methods in Cecil. In *ECOOP'92, Sixth European Conference on Object-Oriented Programming,* p. 33-56, June 1992, Utrecht, The Netherlands. Springer-Verlag, LNCS 615.
7. Chambers, C., D. Ungar, Making Pure Object-Oriented Languages Practical. In *OOPSLA'91, Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, p. 1-15, Phoenix, Arizona, Oct. 1991.
8. Chambers, C., D. Ungar, and E. Lee, An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *Lisp and Symbolic Computation* 4(3), p. 243-281, Kluwer Academic Publishers, June 1991. (Originally published in Proc. *OOPSLA'89).*
9. 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, Jan. 1977.
10. Graver, J.O., *Type-Checking and Type-Inference for Object-Oriented Programming Languages*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989.
11. Graver, J.O. and R.E. Johnson, A Type System for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, p. 136-150, San Francisco, California, Jan. 1990.
12. 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.
13. Hölzle, U., Why Static Typing is not Important for Efficiency. In: J. Palsberg & M.I. Schwartzbach (eds.) *Types, Inheritance, and Assignment,* Technical Report, Daimi PB-357, Computer Science Department, Aarhus University, Denmark, June 1991.
14. Hölzle, U. and D. Ungar, Optimizing Dynamically-Dispatched Calls with Run-time Type Feedback. In *PLDI'94, Conference on Programming Language Design and Implementation,* p. 326-336, Orlando, Florida, June 1994.
15. Johnson, R.E., Type-Checking Smalltalk. In *OOPSLA'86 Object-Oriented Programming Systems, Languages and Applications,* p. 315-321, Portland, Oregon, Sept. 1986.

16. Milner, R., A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, p. 348-375, 1978.
17. 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.
18. Palsberg, J. and M.I. Schwartzbach, Object-Oriented Type Inference. In *OOPSLA'91 Object-Oriented Programming Systems, Languages and Applications*, p. 146-161, Phoenix, Arizona, Oct. 1991.
19. Phillips, G. and T. Shepard, *Static Typing Without Explicit Types*. Technical Report, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, 1994.
20. 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, Oct. 1994.
21. Plevyak, J., X. Zhang, and A.A. Chien, Obtaining Sequential Efficiency in Concurrent Object-Oriented Programs. In *Conference Record of the 22nd Symposium on Principles of Programming Languages*, p. 311-321, San Francisco, California, Jan. 1995
22. Sanella, M., J. Maloney, B. Freeman-Benson, 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.
23. 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, Jan. 1981.
24. Ungar, D. and R.B. Smith, SELF: The Power of Simplicity. *Lisp and Symbolic Computing*, 4(3), p. 187-205, Kluwer Academic Publishers, June 1991. (Originally published in Proc. *OOPSLA'87).*
25. Vitek, J., N. Horspool, and J.S. Uhl, Compile-Time Analysis of Object-Oriented Programs. In *Compiler Construction 4th International Conference, CC'92*, p. 236-250, Paderborn, Germany, Oct. 1992, Springer-Verlag LNCS 641.

Appendix

This appendix discusses issues that we have encountered during three years of work on type inference. Tackling them is important to obtain good type inference results, but since they are not central to the understanding of CPA, we only discuss them briefly in this appendix. Most issues are not specific to CPA, nor to Self.

Incrementality. Fast type inference is essential for some applications, e.g., to support browsing during program development. A large speedup can be achieved by reusing templates from one analysis to the next. If a method in the target program is modified, its templates are flushed, but all other templates are kept. Reanalyzing the program is fast because most of the needed templates already exist. Speedups can also be observed *across* programs where shared code often needs to be analyzed only once. Take arithmetic (bigInt's etc.). The first program that uses arithmetic forces analysis of the methods implementing arithmetic. Subsequent programs can be analyzed faster because they can be connected to arithmetic templates that were created during the first analysis. Incremental analysis may be less precise than a full analysis, unless care is taken to avoid that assignments to global state are visible across analyses.

Blocks, closures, and precision. Expressions in blocks may access slots in their lexical environment. To retain overall precision, such accesses must be analyzed precisely. Both blocks in the `max:` method access slots in the lexical environment:

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

Let a_{slot} denote the slot `a`, and a_{read} the expression which reads the slot `a` in the block `[a]`. Let $type_T$ be the type of a slot or an expression in a template T. Assume we have two `max:`-templates, S and T, for which:

$type_S(\texttt{self}) = type_S(a_{slot}) = \{\overline{\texttt{int}}\}$ and
$type_T(\texttt{self}) = type_T(a_{slot}) = \{\overline{\texttt{float}}\}$.

What should $type(a_{read})$ be? A simple approach is to assume that a_{read} may access a_{slot} in *any* `max:`-template., i.e., $type(a_{read}) = type_S(a_{slot}) \cup type_T(a_{slot}) = \{\overline{\texttt{int}}, \overline{\texttt{float}}\}$. Unfortunately, precise types cannot be inferred under this pessimistic assumption. To do better, analysis must closely parallel execution. Invoking `max:` with an integer argument creates an activation record in which `a` is bound to an integer. When the block `[a]` is evaluated, the result is a closure whose lexical pointer is set to this activation record. Later, when the closure is invoked, the lexical pointer is used to retrieve the value of `a`. To improve type inference precision, this two-phase evaluation of blocks must be simulated. Templates correspond to activation records, so we create type-inference-time closures that have a lexical pointer referring to a template, e.g., (`[a]`, S). When the closure is analyzed, the lexical pointer is used to find, e.g., that $type(a_{read}) = type_S(a_{slot}) = \{\overline{\texttt{int}}\}$. Now it is possible to infer precise types.

The above approach is not specific to Self- or Smalltalk style blocks; e.g., it also applies to Pascal's nested procedures, Scheme's closures and Beta's nested patterns.

Closures and Termination. The precise way to analyze blocks may result in non-termination of type inference, when using an adaptive type inference algorithm such as the iterative algorithm or CPA. The reason is that new templates result in creation of new block closures which in turn may result in creation of new templates. The simplest example is a method which invokes itself with a closure of its own:

```
foo: blk = ( foo: [blk value] ).
```

More complicated cases involve indirect recursion. Plevyak and Chien discuss this in [20]. The general solution is to break the infinite computation by creating a cycle in the constraint graph.

Closures and efficiency. Efficiency of type inference is highly sensitive to the number of closures created. This is due to "self-amplification": more closures result in more templates which in turn result in more closures etc. Consequently, we should create as few closures as possible. However, as outlined above, the simple approach of having only a single closure per block is too imprecise. One way to reduce the number of closures, without impairing precision, is to "fuse" closures when this is safe. Assume we have four templates for `max:`, S and T from above, and U and V with

$type_U(\texttt{self}) = \{\overline{\texttt{int}}\}$, $type_U(a_{slot}) = \{\overline{\texttt{float}}\}$ and
$type_V(\texttt{self}) = \{\overline{\texttt{float}}\}$, $type_V(a_{slot}) = \{\overline{\texttt{int}}\}$.

Even though we have four `max:`-templates, two closures for each of the blocks `[self]` and `[a]` suffice. The reason is that the types in the `[a]` block only depends

on the type of a_{slot}. This type is the same in S and V, and it is the same in T and U. Creating two closures ([a], {S,V}) and ([a], {T,U}) saves analysis effort because it, in turn, will necessitate fewer `ifTrue:False:`-templates. This optimization complicates matters because closures at analysis time, unlike at execution time, now may have several scopes. For Self, we found the performance gain high enough to be worth the added complexity.

Reaching definitions. Assignments are non-destructive in type inference. They do not erase the old type of a slot they assign to, so clone families keep accumulating in the types for assignable slots. This conservatism is necessary in general, but sometimes we can do better. Four increasingly powerful approaches are: do nothing, eliminate dead initializers, use static single assignment (SSA) form, and propagate definitions through the call graph as it is constructed during type inference. In the Self implementation we have tested the first three approaches, but only for local slots. ("Instance slots" are harder to deal with because they may both be aliased and represent multiple locations). We found that the simple elimination of unused initializers pays off significantly, and that SSA is only slightly better. We suspect that this is because most local slots are initialized to `nil`, but immediately assigned to something more useful; simply getting rid of $\overline{\texttt{nil}}$ from the type of the slot achieves most of what can be obtained.

Coupled arguments. Assume that type(x) = {$\overline{\texttt{int}}$, $\overline{\texttt{float}}$} and type(y) = {$\overline{\texttt{int}}$, $\overline{\texttt{float}}$}. The send `x+y` connects to four templates, one for each of the four tuples in the cartesian product of the types of `x` and `y`. In contrast, the send `x+x` should only connect to two templates: ($\overline{\texttt{int}}$, $\overline{\texttt{int}}$) and ($\overline{\texttt{float}}$, $\overline{\texttt{float}}$), since if the receiver is an integer, so is the argument (the form of the send enforces that the receiver and argument is the same object). The receiver and argument are *coupled.* Coupling need not involve the exact same object, just objects that are (somehow) bound to be in the same clone family. In general, determining if two arguments are coupled is not computable, but safe approximations are possible. We implemented a simple approximation, but it had only a marginal effect on analysis of typical Self programs, possibly because we need more powerful detection, but more likely just because coupling *is* rare.

Grouping. The Self type inferencer analyzes an image of objects, not programmer-written source code. Consequently, it may encounter many "similar" objects in the image (e.g., 10,000 point objects that are part of a graph on the display). Inferring types for each object individually is computationally infeasible. To overcome this problem, we group objects according to structural similarity before inferring types [4]. Grouping may cost precision, but speeds up inference since all members of a group can be analyzed "in parallel."