Demand-Driven Type Inference with Subgoal Pruning: Trading Precision for Scalability

S. Alexander Spoon and Olin Shivers

Georgia Institute of Technology

Abstract. After two decades of effort, type inference for dynamically typed languages scales to programs of a few tens of thousands of lines of code, but no further. For larger programs, this paper proposes using a kind of demand-driven analysis where the number of active goals is carefully restricted. To achieve this restriction, the algorithm occasionally *prunes* goals by giving them solutions that are trivially true and thus require no further subgoals to be solved; the previous subgoals of a newly pruned goal may often be discarded from consideration, reducing the total number of active goals. A specific algorithm **DDP** is described which uses this approach. An experiment on **DDP** shows that it infers precise types for roughly 30% to 45% of the variables in a program with hundreds of thousands of lines; the percentage varies with the choice of *pruning threshold*, a parameter of the algorithm. The time required varies from an average of one-tenth of one second per variable to an unknown maximum, again depending on the pruning threshold. These data suggest that 50 and 2000 are both good choices of pruning threshold, depending on whether speed or precision is more important.

1 Introduction

While dynamic programming languages have many advantages, *e.g.*, supporting productive environments such as Smalltalk [1, 2] and the Lisp Machine [3], they share the fundamental weakness that less information about programs is immediately apparent before the program runs. This lack of information affects both programmers and their tools. Programmers require more time tracing through a program in order to answer questions such as "what kind of object is the 'schema' parameter of this method?" Tools such as refactoring browsers [4] and compilers [5, 6] are similarly hampered by the lack of type information. Refactoring browsers cannot make as many safe refactorings, and compilers cannot optimize as much.

To counteract this lack of information, there have been a number of attempts to *infer* types as often as possible for programs written in dynamically typed languages [6–11]. The algorithms from this body of work are quite successful for programs with up to tens of thousands of lines of code, but do not scale to larger programs with hundreds of thousands of lines.

For analyzing programs with hundreds of thousands of lines, this paper proposes modifying existing algorithms in two ways: (1) make them *demand-driven*, and (2) make them *prune subgoals*. The rest of this paper describes this approach in more detail, describing a specific algorithm **DDP** which instantiates the approach, and giving experimental results showing **DDP**'s effectiveness.

2 Previous Work

2.1 Type Inference in Dynamic Languages

The type-inference problem is similar for various dynamic languages, including Smalltalk, Self, Scheme, Lisp, and Cecil. Algorithms that are effective in one language are effective in the others. As pointed out by Shivers [12], all of these languages share the difficulty that the analysis of control and data flow is interdependent; all of these language have and use dynamic dispatch on types and have data-dependent control flow induced by objects or by higher-order functions.

Efforts on this problem date back at least two decades. Suzuki's work in 1981 is the oldest published work that infers types in Smalltalk without any type declarations [7]. More recently, in 1991, Shivers identified *context selection*, called *contour selection* in his description, as a central technique and a central design difference among type-inference algorithms [6]. The core of Agesen's "Cartesian Products Algorithm" (**CPA**) is the insight of choosing contexts in a type-specific way [9]. **CPA** selects contexts as tuples of parameter types; the algorithm gets its name because the tuples are chosen as cartesian products of the possible argument types for each method. More recently yet, Flanagan and Felleisen have increased the speed and scalability of type-inference algorithms by isolating part of the algorithm to work on individual modules; thus, their algorithm spends less time analyzing the entire program [10]. Most recently of all, Garau has developed a type inference algorithm for a subset of Smalltalk [11]. However, the algorithm does not accurately model two important features of Smalltalk: multiple variables may refer to the same object, and objects may reference each other in cycles. Thus Garau's algorithm supports a much smaller subset of the full language than is usual, and it cannot be directly compared to other work in this area.

Unfortunately, the existing algorithms do not scale to dynamic programs with hundreds of thousands of lines. Certainly, no successful results have been reported for such large programs. Grove, *et al.*, implemented a wide selection of these algorithms as part of the Vortex project, and they report them to require an unreasonable amount of time and memory for larger Cecil programs. [5] Their experimental results for analyzing Cecil are summarized in Table 1. Clearly, **0-CFA** is the only algorithm of these that has any hope for larger programs; the various context-sensitive algorithms tend to fail even on programs with twenty thousand lines of code. However, even **0-CFA**, a context-insensitive algorithm that is faster but less precise, seems to require an additional order of magnitude of improvement to be practical for most applications. With the single order of magnitude in speed available today over the test machine of this study (a Sun

	b-CPA	SCS	0-CFA	1,0-CFA	1,1-CFA	2,2-CFA	3,3-CFA
richards (0.4 klocs)	4 sec 1.6 MB	3 sec 1.6 MB	3 sec 1.6 MB	4 sec 1.6 MB	5 sec 1.6 MB	5 sec 1.6 MB	4 sec 1.6 MB
deltablue (0.65 klocs)	8 sec 1.6 MB	7 sec 1.6 MB	5 sec 1.6 MB	6 sec 1.6 MB	6 sec 1.6 MB	8 sec 1.6 MB	10 sec 1.6 MB
instr sched (2.0 klocs)	146 sec 14.8 MB	83 sec 9.6 MB	67 sec 5.7 MB	99 sec 9.6 MB	109 sec 9.6 MB	334 sec 9.6 MB	1,795 sec 21.0 MB
typechecker (20.0 klocs)	∞ ∞	∞ ∞	947 sec 45.1 MB	13,254 sec 97.4 MB	∞ ∞	∞ ∞	∞ ∞
new-tc (23.5 klocs)	∞ ∞	∞ ∞	1,193 sec 62.1 MB	9,942 sec 115.4 MB	∞ ∞	∞ ∞	∞ ∞
compiler (50.0 klocs)	∞ ∞	∞ ∞	11,941 sec 202.1 MB	∞ ∞	∞ ∞	∞ ∞	∞ ∞

Table 1. Type-inference performance data from Grove, *et al.* [5]. Each box gives the running time and the amount of heap consumed for one algorithm applied to one program. A ∞ entry means attempted executions that did not complete in 24 hours on the test machine.

Ultra 1 running at 170 MHz), and with the cubic algorithmic complexity of **0-CFA**[13], one can expect a program with one hundred thousand lines to be analyzed in about 2.7 hours and to require 1600 MB of memory. A program with two hundred thousand lines would require 21 hours and 13 GB of memory. While one might quibble over the precise definition of "scalable", **0-CFA** is at best near the limit. Grove, *et al.*, agree with this assessment:

> The analysis times and memory requirements for performing the various interprocedurally flow-sensitive algorithms on the larger Cecil programs strongly suggest that the algorithms do not scale to realistically sized programs written in a language like Cecil [5].

2.2 Concrete Type Inference in Static Languages

Type inference (by which we mean "concrete type inference" or "class analysis") is a much easier problem in languages like Java and C which have static type information available, and there are already scalable analyses for these languages. It is not a trivial problem, because better types may be inferred than the type system provides, but it is nevertheless a much easier problem.

One way the problem is easier is that context sensitivity is not necessary. For example, Tip and Palsberg have developed an effective analysis that does not need context [14]. In fact, Tip and Palsberg argue that existing context-sensitive algorithms in Java should be abandoned as impractical:

> In practice, the scalability of the algorithms at either end of the spectrum is fairly clear. The CHA and RTA algorithms at the low end

of the range scale well and are widely used. The k-CFA algorithms (for $k > 0$) at the high end seem not to scale well at all.[14]

It appears that the static types in static languages are helping by bounding the distance that an imprecise type may flow. For some intuition about this effect, consider the following example. Suppose we define an identity method, `yourself`, in Smalltalk—a simple polymorphic method that returns whatever is passed to it. Consider this use of the `yourself` message:

```
x := x yourself
```

This statement has no effect on the value of x, much less its type. With a context-free analysis, however, every invocation of `yourself` must be assigned the same result type, and thus the presence of this statement causes the type of x to be polluted by the types of every other sender of `yourself` in the entire program. Contrast this result to a context-sensitive analysis. With **CPA**, the `yourself` method is analyzed multiple times, once for each type of argument supplied to it. With **1-CFA** [6], the method is analyzed once for each place it is called from. In either case, the type of x will remain the same whether this statement is present or not.

To contrast, consider the same code converted to Java, where `yourself()` is an identity method and `Foo` is some class:

```
Object yourself(Object o) {return o;}
...
Foo x;
...
x = (Foo) yourself(x)
```

In this case, any context-free analysis is sufficient. The type of the expression `yourself(`x`)` will still be polluted just as in Smalltalk, but the cast will protect the type of x itself from being polluted. The statement may cause x's type to become less precise, but it will not become less precise than `Foo`.

The example generalizes well beyond identity methods. This situation arises with *any* polymorphic method that is called by monomorphic code. In Java, the monomorphic code will still get precise types, because there will always be casts to keep the imprecise types of the polymorphic code from polluting the monomorphic code. In dynamic languages there is no such protection, and only the use of context can keep the imprecise types at bay. Unfortunately, as described in the previous section, the known context-sensitive algorithms do not scale to hundred-thousand line programs in dynamically typed languages.

3 A Scalable Approach

To scale type inference to larger programs, this paper proposes two techniques: the algorithms are *demand-driven*, and they *prune subgoals*. Each technique is described below.

3.1 Demand-Driven Analysis

Demand-driven algorithms find information "on demand:" instead of finding information about every construct in an entire program, they find information that is specifically requested. Several demand-driven versions of data-flow algorithms have been developed [15–19].

Demand-driven algorithms are organized around *goals*. A client posts *goals* that the algorithm is to solve, *e.g.*, "what is the type of x?" To find a correct response to a goal, the algorithm may itself post subgoals. For example, in response to the query "what is the type of x?", it may ask "what is the type of y?" if there is a statement "x := y" in the program. As the algorithm progresses, it has a number of goals it is trying to solve; when it finishes, it has solved some number of those goals.

There are two main advantages to using demand-driven analysis instead of the usual exhaustive analysis. First, a demand-driven algorithm analyzes a subset of the program for each goal. For a particular goal, often only a small number of subgoals are needed and only a limited portion of the program is analyzed. Thus a demand-driven algorithm typically has much less work to do and can finish each execution much more quickly than an entire exhaustive analysis can complete. Since it is frequently the case that program-manipulation tools such as compilers and development environments only sparsely probe the program with queries, having an algorithm that doesn't require exhaustively computing the answer to *every* query just to provide answers for *some* queries can pay off handsomely.

Second, demand-driven algorithms can adaptively trade off between precision of results and speed of execution. If the algorithm completes quickly, then it can try more ambitious subgoals that would lead to more precise information about the target goal. Likewise, if the algorithm is taking too long, it can give up on some subgoals and accept less precise information. The latter idea is explored in Sect. 3.2.

The main disadvantage of a demand-driven analysis is that it only finds information about those constructs for which goals have been posted. If an application *does* need the answers for all possible queries of a given program, then an exhaustive algorithm (if one is available) can be faster than running the demand-driven algorithm once for each query, as the exhaustive algorithm may be able to share information across the various queries being processed.

3.2 Subgoal Pruning

A goal is pruned by giving it an immediate solution which is correct (albeit probably lacking in precision) independent of the answer to any other goals. In other words, a pruned goal has no subgoals. By carefully choosing goals to prune, an algorithm may reduce the number of goals that are relevant to the target goal. In turn, having fewer goals tends to speed up the algorithm. Thus, pruning subgoals is a technique for trading precision for speed.

Pruning subgoals is not itself a new idea. It is an instance of *heuristic search*, a well-known technique of artificial intelligence [20]. However, the technique is rare in published program analyses. Published work tends to insist on optimal solutions within some rules of inference, even though these rules are themselves typically approximate, *e.g.*, data-flow analyses include flow paths in their meet-over-paths specification that may never occur at run time. Some researchers have designed clever techniques that adaptively *increase* the number of goals to gain more precision when time allows [19]. We know of no published work in program analysis that adaptively *reduces* the number of goals in response to resource limitations.

3.3 Synthesis

The combination of being demand-driven and of adaptively pruning subgoals yields a nice synthesis where precise information is found in two different situations. First, precise information is found whenever ambitious context-sensitive expansion of subgoals is both effective and inexpensive. Note that just because context-sensitive expansion *allows* a huge number of subgoals to be generated, the expansion does not *always* do so. Second, precise information is found whenever context is not necessary. Each goal being analyzed gets both opportunities, and different subgoals can use different opportunities within a single execution of the analysis. Intuitively, the synthesis might be described as: *aim high but tolerate less*. For the algorithm designer, the benefit is that context-sensitive justification rules can be added without much worry about the worst cases of subgoal explosion.

4 The DDP algorithm

We have designed an algorithm using these principles to perform type analysis of programs written in a dynamically-typed object-oriented language such as Smalltalk. This algorithm is named **DDP**, because it is demand-driven and it prunes subgoals.

DDP searches backwards from a root goal g, which is the original request. We essentially build a proof tree, in a goal-directed manner, using Agesen's CPA abstraction. This abstraction provides for an analytic space that is finite, albeit intractably large for large programs. However, let us assume for the moment that we have unbounded computational resources at hand for this search (we'll fix this assumption later). Were this the case, we could explore the proof tree backwards, terminating the search at (1) axioms (that is, leaf nodes), and (2) circular dependencies. An example of an axiom would be an assertion that the type of the value produced by sending a `new:` message to a class C in some context has type $\{C\}$. An example of a circular dependency would be found by searching backwards through a loop or recursion. The presence of circular dependencies means that our proof tree is actually a rooted graph, of course. Note that this proof graph, for a particular request or root goal, might quite likely make reference to only a subset of the entire program under consideration. (That, at least, is our hope.)

Once this proof graph had been determined by backwards search, we would then propagate information *forwards* from the axioms. Circular dependencies in the graph are handled by iterating to convergence. However, flowing information forwards may trigger the creation of further subgoals in our graph. This is in the nature of higher-order control-flow analysis: as we discover values of new types flowing to the receiver position of a message-send statement, determining the value produced by the entire statement will require establishing new subgoals for message sends to objects of the newly-discovered classes for the receiver. Thus backwards search and forwards information flow are interleaved.

The facts that concern our proof graph (type and control-flow assertions) have a lattice structure; a fixed point is a valid solution (which is how we account for cycles in the proof graph). By valid, we mean that any possibility that might arise during the execution of the program is described by the asserted type or flow fact. Note that even the least fixed-point solution to the proof graph might also include possibilities that never arise during actual program execution; this fundamental limitation is typical for this kind of analysis. Further, any value above the least fixed point is also valid, in that it will be a less precise approximation admitting all the possibilities of the least fixed-point solution, and others as well.

With this structure in mind, let's now adjust the algorithm to account for our need to do analysis in the presence of bounded computational resources. As we search backwards, recursively expanding out the proof graph, we associate with each node in our proof graph a current value from our fact lattice. For a newly-created goal, this fact starts out as the bottom fact. At all times, this value is $\sqsubseteq$ the fixed-point value we would compute if we had unbounded resources. That is, we are always consistent with the fixed-point value, but possibly more precise than it is (*i.e.*, *too* precise)—the current value at a node might not account for all the cases that the final fixed-point value would cover. As our search uncovers new sources of information, we flow this information forward through the graph.

If we are fortunate enough for this entire process to settle out and terminate within some previously established resource budget, then we can halt with a fixed-point solution that provides our final answer. If, however, we exhaust time or space, then we may choose to prune subgoals. This is a standard technique from the AI community, which has long dealt with the problem of search in intractably large spaces. We can choose a fringe of unsatisfied nodes in the tree, and simply assert the top fact for these nodes—the fact that is so pessimistically or conservatively general that it *must* cover the actual run-time behavior described by the goal in question. Hopefully, if this fact occurs deep in the search tree, it will not have too large an effect on the precision of the fact produced at the root of the tree.

Introducing top values in place of more precise facts to elide search means that our final result must now be above the least fixed-point solution we would have found with our simple, unbounded-resource algorithm. So this is where we introduce approximation to handle resource bounds on the analysis.

Note that the bidirectionality of the analysis means that simply trimming the search tree and asserting top facts does not give us an immediate solution. Recall that forward flow of facts that are high in the lattice may require us to create new subgoals higher in the search tree. However, taking a large step upwards in the fact lattice certainly moves us closer to an eventual fixed point.

This, in a nutshell, is the essence of our algorithm. The rest is simply filling in the particulars of our particular analysis problem (OO type inference) and finite abstraction (Agesen's CPA).

4.1 Overall Algorithm

```
procedure InferType(var) {
  rootgoal := typegoal(var)
  worklist := { rootgoal }

  while worklist ≠ ∅ do
    if resource bounds unexhausted
    then Search1()
    else Prune()
}

procedure Search1() {
  Remove g from worklist
  changed? := Update(g)

  if changed? then
    deps := Depends_on(g)
    worklist := worklist ∪ deps
}

procedure Prune() {
  for g ∈ ChoosePrunes() do
    prune g
  worklist := Relevant(rootgoal)
}
```

Fig. 1. The top-level structure of the **DDP** algorithm.

The top-level structure of the algorithm is shown in Figure 1. The algorithm is based on a worklist; it commences by initializing the worklist to the initial top-level goal. The worklist-processing loop, however, is sensitive to resource usage.

As long as resources (either time or space) have not been exhausted, the basic loop removes a work item g from the worklist and processes it. Each work item corresponds to a node in the proof graph we are constructing. Each such node has associated with it a current tentative solution in the fact lattice, the set of nodes on which it is dependent, and the set of nodes that depend on it. First, we update g, recalculating its value in the fact lattice from the current nodes in the proof graph upon which it is dependent (change in these nodes was the reason for g being added to the worklist). If the justification requires subgoals that do not exist, then new goals are created and added to the worklist. New goals are always given a maximally optimistic tentative solution, *e.g.*, $\perp$ for type goals. Once all the necessary subgoals have been created or located, and g's solution is consistent with all its subgoals, we may remove g from the worklist and proceed. However, if any of this processing has caused g's current value to rise in the fact lattice, then any goal depending on g must be revisited. These goals are added to the worklist. In this way, the graph is searched backwards, in a goal-directed fashion, and values are propagated forwards through this graph; when we arrive at a fixed point, the worklist will be exhausted.

Once the search has proceeded beyond pre-set resource limits, however, a pruning step is triggered. The pruning step selects some number of nodes to prune, assigns to each one the appropriate top value, and performs a graph search starting at the root goal in order to find the goals that are still relevant. Those goals found in the graph search become the new contents of the worklist.

4.2 Mini-Smalltalk

DDP analyzes full Smalltalk, but many of the details are more tedious than interesting. In this paper, **DDP** is described as an analysis of Mini-Smalltalk, a minimal language that captures the essence of Smalltalk.

A program in Mini-Smalltalk has a list of global variables and a class hierarchy with single inheritance. Each class has a list of instance variables and a list of methods. Each method has a selector and a main block. A block, including the main block of a method, has a list of parameters, a list of local variables, and a list of statements.

A statement in Mini-Smalltalk is of one of the following forms:

- *var* := *literal*. This statement assigns a literal to the specified variable. The precise set of literals is unimportant, except to note that method selectors are valid literals.
- *var* := *rvar*. This statement assigns one variable to another.
- *var* := `self`. This statement assigns the current receiver to the variable *var*.
- *var* := `new` *class*. This statement creates a new object with the specified class, and it sets all of the new object's instance variables to `nil`.
- *var* := `send`(*rcvrvar*, *selector*, $argvar_1$, …, $argvar_m$). This statement initiates a message send. When the invoked method completes, the result will be stored into *var*.

- *var* := `sendvar`(*rcvrvar*, *selvar*, $argvar_1$, ..., $argvar_m$). This statement also invokes a method, with the difference that the selector is read from a variable instead of being specified in the syntax. In Smalltalk, this functionality is invoked with the `perform:` family of methods.
- `return` *var*. This statement returns a value to the calling method.
- *var* := *block*. This statement creates a new block, which has its own parameters, local variables, and statements, and assigns that block to a variable. A block statement is the same as a lambda expression in a functional language. A block captures any variable bindings in its surrounding lexical scope, it can be passed around the program as a first-class value, and it can be evaluated later.
- *var* := `beval`(*blockvar*, $argvar_1$, ..., $argvar_m$). This statement reads a block from a variable and invokes the block with the specified parameters.
- `breturn` *var*. This statement returns a value from a block. Note that `breturn` and `return` are not the same: the latter will return from the whole method, and thus may simultaneously pop multiple levels of the runtime call stack.

It is assumed that all variable names in a Mini-Smalltalk program are distinct. Since variables are statically bound in Smalltalk, this causes no loss of generality.

Most program constructs in full Smalltalk can be translated to Mini-Smalltalk. Variables may be renamed to be unique because all variable references are statically bound. A compound expression may be rewritten as a series of simple statements that assign subexpressions to fresh temporary variables. The primitive `value` method for evaluating a block may be replaced by a single Mini-Smalltalk method that executes a `beval` on `self`. Likewise the `perform:` method can be replaced using a `sendvar` statement. Note that the Mini-Smalltalk `new` statement is not intended to model the full effect of sending a `new` message to some class; that is handled by the full body of the class's corresponding method. The Mini-Smalltalk `new` statement simply provides for the primitive allocation part of such a computation.

A few items are missing from Mini-Smalltalk, but cause no fundamental difficulty if they are added back. The `super` keyword is missing, but if `super` message sends are added back, they can be treated in the same way normal message sends are treated, only with a different and simpler method lookup algorithm. Primitive methods are missing, but if they are added, a type inferencer can analyze most of them in a straightforward fashion so long as a few descriptive functions are available for the primitives of interest. These descriptive functions give information such as what return type the primitive has when it is supplied with various input types. Primitives without descriptive functions may still be analyzed soundly by using conservative approximations.

There are remaining primitives in full Smalltalk which cause difficulties. For example, one such primitive, analogously to the `eval` function of Lisp, creates a new method from an arbitrary list of bytecodes. Such methods are inherently very difficult for static analysis. However, it is quite rare to see these capabilities used outside of program-development environments. Most applications stay away from such powerful and difficult-to-analyze reflection facilities: they can make programs hard for humans to understand, for the same reasons that they make them difficult for compilers to analyze and optimize. Thus a static analysis is still useful if it ignores such constructs, and in fact ignoring such constructs is par for most static analyses to date.

4.3 Goals

The algorithm has five kinds of goals. Informally they are:

- type goals, *e.g.*, "what is the type of x?"
- transitive flow goals, *e.g.*, "where can objects flow, if they start in x?"
- simple flow goals, *e.g.*, "where can objects flow after just one program step, if they start in x?"
- senders goals, *e.g.*, "what statements can invoke the method named + in class SmallInteger?"
- responders goals, *e.g.*, "what methods are invoked by [x := 2 + 2] ?"

Each goal is answered by its own kind of *judgement*. For example, the type goal "what is the type of x" could be answered by a type judgement "x is of type Integer."

This section will describe these kinds of goals and judgements in more detail, but first it is necessary to describe *types*, *contexts*, and *flow positions*.

Types A *type* describes a set of objects. The following kinds of types are available:

- $\{c\}$, a *class type* including all instances of class c. A class type does *not* include instances of subclasses.
- $S\{sel\}$, a *selector type* including the sole instance of the method selector sel. (Selector types are included in the list so that `sendvar` statements can be handled.)
- $B\{blk\}_{ctx}$, a *block type*, including all block objects which were created from a block statement $[v := blk]$ in context ctx (contexts are described below).
- $t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n$, a union of two or more types of the above kinds.
- $\top$, a type including every object.
- $\bot$, a type including no objects.

As usual, one type may be a subtype of another, $t_1 \sqsubseteq t_2$, if all objects in t_1 are also in t_2. For example, every type is a subtype of $\top$. Further, the union of two types, $t_1 \sqcup t_2$, is the set of objects included in either t_1 or t_2. Note, that subclasses are not included in class types, *e.g.*, {Object} is *not* a supertype of {Integer}.

Contexts A *context* specifies a subset of possible execution states. It does so by specifying a block or method and by placing restrictions on the types of the parameters of the current method or block, and possibly by restricting the type of the current message receiver. A valid context may only restrict parameters that are visible within the specified block. A valid context cannot, for example, restrict the types of parameters from two different methods (as methods do not lexically nest). Further restrictions on valid contexts are described further below.

Most contexts are written as follows:

$$<: (\textbf{Foo.addBar:})\ \texttt{self} = \{\text{Foo}\},\ \ \texttt{x} = \{\text{Bar}\} :>$$

This context matches execution states for which all of the following are true:

- The block that is executing is either the main block of the method named `addBar:` in class Foo, or is a block nested within that method.
- The receiver, *i.e.*, `self` in Smalltalk, is a member of type {Foo}.
- The parameter named `x` holds an object of type {Bar}.

Additionally, the context $\top_{ctx}$ matches any execution state, and the context $\bot_{ctx}$ matches no execution state.

Like types, contexts may be compared with each other; $ctx_1 \sqsubseteq ctx_2$ if every state matched by ctx_1 is also matched by ctx_2. Furthermore, $ctx_1 \sqcap ctx_2$ is a context which matches a state if the both ctx_1 and ctx_2 also match it. Likewise, $ctx_1 \sqcup ctx_2$ matches states that either ctx_1 or ctx_2 match.

There is a mutual recursion between the definitions of types and contexts, and it is desirable that only a finite number of types and contexts be available. Otherwise, the algorithm might have what Agesen terms "recursive customization" [9]. To gain this property, a technical restriction is imposed: the context of a block type $B\{blk\}_{ctx}$ is not allowed to mention, either directly or indirectly, another block type whose block is *blk*. The restriction is enforced whenever the algorithm is about to create a new block type: a recursive traversal of the context is performed and any context mentioning *blk* is replaced by $\top_{ctx}$. While this approach is surely suboptimal, it does not appear to be an issue in practice.

Flow Positions A *flow position* describes a set of locations in the program that can hold a value. The following kinds of flow positions are used by **DDP**:

- $[: V\ var :]_{ctx}$ is a *variable flow position*, and it describes the variable *var* while execution is in context *ctx*.
- $[: S\ meth :]_{ctx}$ is a *self flow position*, and it describes the current receiver while the method *meth* is executing under the context *ctx*.
- $fp_1 \sqcup fp_2 \sqcup \cdots \sqcup fp_n$ is a union of a finite number of flow positions, and it describes the locations described by any of $fp_1, \ldots, fp_n$. Each fp_i must be a flow position of one of the above kinds.
- $\top_{fp}$ is the universal flow position. It includes every possible location in the program.
- $\bot_{fp}$ is the empty flow position. It includes no locations.

Type Goals A *type goal* asks what the type of some variable is, when the variable appears in some context. It looks like:

$$var :_{ctx} ?$$

The solution to this type goal is some *type judgement* $var :_{ctx} type$. This judgement declares that as the program executes, the variable *var*, whenever it appears in a context matching *ctx*, will only hold objects of type *type*.

Flow Goals A *simple flow goal* asks where values may flow in one program step, given a starting flow position. It looks like:

$$fp_1 \rightarrow ?$$

The solution to a simple flow goal is some *simple flow judgement* $fp_1 \rightarrow fp_2$. This judgement claims that after any one step of execution, an object that was at flow position fp_1 may only now be found in either positions fp_1 and fp_2. fp_1 is a possibility because the judgement claims that the object *may* flow to fp_2, not that it *will.*

A *transitive flow goal* asks the same question except that any number of program steps is allowed. A transitive flow goal looks like:

$$fp_1 \rightarrow^* ?$$

The solution to this transitive flow goal is a *transitive flow judgement* $fp_1 \rightarrow^* fp_2$. This judgement claims that if an object is located only in position fp_1, then after any number of program steps, it may only be found in position fp_2.

Senders Goals A *senders goal* asks what statements may invoke a specified method or block.

$$blk_{ctx} \xleftarrow{send} ?$$

The solution to this senders goal is a *senders judgement* $blk_{ctx} \xleftarrow{send} ss$. This judgement claims that whenever a step of execution will invoke the block *blk*, there must be a statement/context pair $(stat, ctx) \in ss$ such that the invoking statement is *stat* and the invoking execution state is matched by *ctx*.

Alternatively, *ss* may be $\top_s$, in which case the judgement makes no claim about which statements may invoke the block.

Responders Goals A *responders goal* asks what methods or blocks may be invoked by a specified statement. It looks like:

$$stat_{ctx} \xrightarrow{send} ?$$

The solution to this responders goal is a *responders judgement* $stat_{ctx} \xrightarrow{send} rs$. Typically, *rs* is a set of block/context pairs. The judgement claims that whenever *stat* executes in context *ctx*, it will invoke a block b in some context *bctx* such that $(b, bctk) \in rs$. If *stat* is a `send` or `sendvar` statement, then all of the blocks in *rs* will be main blocks of methods. If *stat* is a `beval` statement, then all of the blocks will instead be blocks created by block statements.

Alternatively, *rs* may be $\top_r$, in which case the judgement makes no claim about which blocks might be invoked by *stat*.

Valid Contexts We have already seen that a valid context may only specify the types of parameters that are lexically visible within the block specified by the context. An additional restriction appears in each place that a context is used.

- The context for a variable flow position may only specify a block that declares the variable, or which lexically encloses a block declaring the variable. Note we *don't* mean contexts nested *within* the scope of the variable (*i.e.*, the scope in which the variable is visible), but the contexts within which it is nested itself—these are the allocated contexts that are known to exist when the variable is bound itself.
- The context for a self flow position may only specify the main block of the method.
- The context for a block type may only specify a block that encloses the statement which creates the block.
- The context for a type goal may only specify a block that lexically encloses the variable whose type is to be inferred.

If a context is invalid for its intended usage, it can be broadened until it meets the necessary restrictions. The notation $\lceil ctx \rceil$ denotes a context that is less restrictive than *ctx* and which is valid for the intended purpose (as described above). That intended purpose will be clear from the context of discussion. For example, one might write:

$$var :_{\lceil ctx \rceil} type$$

In this type judgement, the context is a valid context at least as inclusive as *ctx*, and if that context is other than $\top_{ctx}$, then it specifies a block that either declares *var* itself or encloses the block that does.

4.4 Justification Rules

When **DDP** updates a goal (see Sect. 4.1), it assigns a new tentative solution to the goal and references one *justification rule* to justify it. The rules available to **DDP** have been chosen, for the most part, to be as simple as possible. The main choice is that the algorithm uses the kind of context sensitivity that **CPA** does.

Unfortunately, as straightforward as the rules are, they still require 17 pages to describe in the current working document giving the formal definition of **DDP**. With five kinds of judgements and ten kinds of statements, the details simply add up. Thus, instead of listing all of the justification rules, we describe only a few representative rules in this section. The full set are straightforwardly derived from the application of Agesen's **CPA** in a backwards-search context.

Usually, a judgement is justified by accounting for every statement that the program might execute in a program $\mathcal{P}$:

$$\frac{\text{J-ALLSTATS} \quad \forall stat \in \mathbf{statements}(\mathcal{P}) : \quad stat \ \rhd \ j}{\rhd \ j}$$

The notation $\rhd \ j$ claims that judgement j is justified without qualification. The notation $stat \ \rhd \ j$ claims that the judgement j accounts for the possible execution of statement $stat$.

Here is the justification rule for type judgements that is used to account for variable-assignment statements:

$$\frac{\begin{array}{c}\text{T-VAR}\\ \rhd \ v_2 :_{\lceil ctx \rceil} t_2 \\ t_2 \sqsubseteq t\end{array}}{[v_1 := v_2] \ \rhd \ v_1 :_{ctx} t}$$

Informally, this justification rule requires the type of v_1 to be a supertype of that for v_2.

Note that using T-VAR requires a subgoal to be satisfied. Whenever the T-VAR rule is used as part of the justification for a solution to the goal $v_1 :_{ctx} ?$, there is a subgoal generated to also find and justify a solution to the goal $v_2 :_{\lceil ctx \rceil} ?$. The solution to the subgoal will be some judgement $v_2 :_{\lceil ctx \rceil} t_2$ that can be used to satisfy the assumptions in T-VAR. This pattern is general: judgements in the assumption list of a justification rule, correspond to subgoals that **DDP** generates.

A more complex rule is used to show that a type judgement accounts for a message send statement:

$$\frac{\begin{array}{c}\text{T-SEND}\\ stat = [v := \mathtt{send}(v_{rcvr}, sel, v_1, \ldots, v_m)] \\ \rhd \ stat_{ctx} \xrightarrow{send} \{(m_1, c_1), \ldots, (m_p, c_p)\} \\ \forall i \in 1 \ldots p : \forall \ v_{ret} \in \mathbf{ret\_vars}(m_i) : \\ \exists t' : (\rhd \ v_{ret} :_{\lceil c_i \rceil} t')) \wedge (t' \sqsubseteq t)\end{array}}{v := \mathtt{send}(v_{rcvr}, sel, v_1, \ldots, v_m) \ \rhd \ v :_{ctx} t}$$

Informally, this rule finds the methods that may respond to the statement, it finds a type for each return statement in those methods, and then it requires that t be a supertype of all of the returned types.

In more detail, the rule first requires that there is a justified responders judgement $stat_{ctx} \xrightarrow{send} \ldots$. This judgement identifies the methods that can respond to this `send` statement along with the contexts those methods can be invoked under. For each tuple (m_i, c_i) of a responding method and context, the function **ret_vars** is used to find the variables returned by a return statement in m_i. For each such variable v_{ret}, the rule requires that there exists a justified type judgement $v_{ret} :_{\lceil c_i \rceil} t'$ giving a type t' for the object returned. Each such t' must be a subtype of t, the type in the type judgement that is being justified.

Finally, here is the main justification rule is used to find the invokers of a block:

S-BEVAL

$$\frac{\begin{array}{c} stat = [v := \mathtt{beval}(v_{beval}, v_1, \ldots, v_m)] \\ [v_{blk} := blk] \in \mathbf{statements}(\mathcal{P}) \\ \rhd [: V \ v_{blk} :]_{\lceil ctx \rceil} \rightarrow^* f_b \\ f_b \sqcap [: V \ v_{beval} :]_{\top_{ctx}} = [: V \ v_{beval} :]_{bctx_1} \sqcup \cdots \sqcup [: V \ v_{beval} :]_{bctx_p} \\ \forall i \in 1 \ldots p : \quad (stat, bctx_i) \in ss \end{array}}{[v := \mathtt{beval}(v_{beval}, v_1, \ldots, v_m)] \ \rhd \ blk_{ctx} \xleftarrow{send} \ ss}$$

The goal of this justification rule is to justify that a senders judgement accounts for a particular `beval` statement. Informally, this rule requires that the block has been traced forward through the program, and that the senders set accounts for each way (if any) the block can reach the `beval` statement.

In detail, the rule first finds the unique statement $[v_{blk} := blk]$ that creates the block. The rule then requires that there is a justified transitive flow judgement $[: V \ v_{blk} :]_{\lceil ctx \rceil} \rightarrow^* f_b$. This flow judgement claims that the only locations the block can reach are those in the flow position f_b. Next, the rule picks out the portions of f_b which is for variable v_{beval}, the variable the `beval` statement is reading its block from. For each such portion $[: V \ v_{beval} :]_{bctx_i}$, the senders set ss must include an entry for the `beval` statement under context $bctx_i$.

4.5 Subgoal Pruning

A complete algorithm must specify *how* to prune, *when* to prune, *which* goals to prune, and how to *garbage collect* goals that are no longer needed. **DDP** has straightforward choices for all of these.

To prune a goal, **DDP** chooses the appropriate $\top$ value for the goal: $\top$ for type goals, $\top_{fp}$ for flow goals, $\top_r$ for responders goals, and $\top_s$ for senders goals. In any case, the goal is immediately justified without depending on any subgoals.

DDP prunes goals when either many goal updates have happened since the last pruning, or many new goals have been added since the last pruning. Specifically, there is a threshold `Thresh`, and if the sum of the number of new updates plus the number of new goals surpasses that threshold, then **DDP** makes a number of prunings and then garbage collects.

Before choosing which goals to prune, **DDP** chooses which goals to keep: it keeps the first `Thresh` goals it encounters during a breadth-first traversal of the proof graph. The goals it chooses to prune are then the goals which are kept but which have subgoals that are not kept.

Garbage collection is then straightforward: any node not to be kept is removed from consideration.

4.6 Example Execution

Figure 2 gives a short fragment of a program. Suppose the type inferencer tries to infer a type for `x`. A goal is posted for the type of `x`, and after a few updates goals for `h` and `doc` are posted, reaching the goal table shown in Table 2.

```
x := 0
doc := (something complicated)
h := send(doc, documentHeight)
x := h
```

Fig. 2. An example program fragment

Note that each assigned type includes {UndefinedObject}, the type of `nil`, because all variable bindings other than parameters are initialized with `nil`. **DDP** makes no effort to determine whether a variable is used before it is initialized, and thus {UndefinedObject} must be included for correctness.

At this point, the algorithm will spend many updates working on the type of `doc`. Eventually the algorithm will give up on `doc` and prune it, leading to the goal table in Table 3. The type of `doc` has been lost, but now the calculation for the type of `h` may proceed. Since only one method in the program has selector `documentHeight` (let us presume), losing the type of `doc` causes no harm. Suppose the one `documentHeight` method has a single return statement returning variable `height`. Then, after a few more steps, the goal table reaches the state shown in Table 4, and a precise type of `x` has been found.

5 Experimental Evaluation

We have implemented **DDP** and performed an experiment with two objectives: to measure the algorithm's overall performance, and to determine a good pruning threshold for the algorithm. This section describes the experiment in more detail, gives the data it produced, and interprets the results.

DDP was implemented in Squeak 2.8, one version of a single large program written in Smalltalk. This implementation was then used to analyze portions of a combined program with Squeak 2.8 plus **DDP** plus the Refactoring Browser[1] as ported to Squeak.[2] (The Refactoring Browser was included because it is used in the implementation of **DDP**). This combined program has 313,775 non-blank lines of code, 1901 classes, and 46,080 methods. Table 5 summarizes the components of this program that were targeted for analysis.

The implementation was given a series of queries, with the queries ranging over (1) instance variables from one of the above packages, and (2) a pruning threshold between 50 and 5000 goals. We ran our tests on a Linux IA32 system

[1] —http://www.refactory.com/RefactoringBrowser—
[2] —http://minnow.cc.gatech.edu/squeak/227—

Goal Number	Goal	Current Solution	Subgoals
1	x $:\top_{ctx}?$	{SmallInteger} $\sqcup$ {UndefinedObject}	2
2	h $:\top_{ctx}?$	{UndefinedObject}	3
3	doc $:\top_{ctx}?$	{UndefinedObject}	*many. . .*
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 2. Example goal table after a few goal updates

Goal Number	Goal	Current Solution	Subgoals
1	x $:\top_{ctx}?$	{SmallInteger} $\sqcup$ {UndefinedObject}	2
2	h $:\top_{ctx}?$	{UndefinedObject}	3,100
3	doc $:\top_{ctx}?$	$\top$	*none*
100	height $:\top_{ctx}?$	{UndefinedObject}	. . .
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3. Example goal table after pruning goal 3

Goal Number	Goal	Current Solution	Subgoals
1	x $:\top_{ctx}?$	{SmallInteger} $\sqcup$ {UndefinedObject}	2
2	h $:\top_{ctx}?$	{SmallInteger} $\sqcup$ {UndefinedObject}	3,100
3	doc $:\top_{ctx}?$	$\top$	*none*
100	height $:\top_{ctx}?$	{SmallInteger} $\sqcup$ {UndefinedObject}	. . .
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4. Example goal table at finish

Name	Instance variables	Description
rbparse	56	Refactoring browser's parser.
mail	70	Mail reader distributed with Squeak
synth	183	Package for synthesis and manipulation of digital audio
irc	58	Client for IRC networks
browser	23	Smalltalk code browser
interp	207	In-Squeak simulation of the Squeak virtual machine
games	120	Collection of small games for Morphic GUI
sunit	14	User interface to an old version of SUnit
pda	44	Personal digital assistant

Table 5. The components of the program analyzed in the experiment.

with a 2.0 GHz Athlon processor and 512Mb of RAM. The speed of the inferencer is summarized in Table 6, and the precision of the inferencer is summarized in Table 7.

To compute the precision percentages, each type inferred for a variable in a given component was hand classified as either *precise* or *imprecise*. We hand classified results using the following rules:

- $\top$ is imprecise.
- $\bot$ is precise. ($\bot$ means that the variable is never instantiated at all.)
- Any simple class type, selector type, or block type is precise.
- A union type is precise if, according to human analysis, at least half of its component simple types may arise during execution. For this analysis, the exact values of arithmetic operations are not considered; *e.g.*, any operation might return a negative or positive result, and any integer operation might overflow the bounds of SmallInteger.
- If none of the above rules apply, then the type is imprecise.

There is clearly some subjectivity in the fourth rule. However, we believe that we have been at least as conservative as any reasonable experimenter would be; when in doubt, we classified a type as imprecise. Further, the following types, which are obviously precise, comprise 87.0% of the inferences that were classified as precise:

- (56.3%) $\{C\} \sqcup \{\text{UndefinedObject}\}$, for some class C.
- (13.7%) $\{\text{UndefinedObject}\}$, *i.e.*, the variable is never initialized from the code.
- (10.3%) $\{\text{True}\} \sqcup \{\text{False}\} \sqcup \{\text{UndefinedObject}\}$
- (6.7%) $\{\text{SmallInt}\} \sqcup \{\text{LargePosInt}\} \sqcup \{\text{LargeNegInt}\} \sqcup \{\text{UndefObject}\}$[3]

[3] Class names have been abbreviated.

	50 nodes	150 nodes	500 nodes	1000 nodes	1500 nodes	2000 nodes	2500 nodes	3000 nodes	4000 nodes	5000 nodes
rbparse	0.16	0.57	2.76	5.57	15.43	16.4	25.33	26.02	33.94	61.33
mail	0.13	0.72	3.12	9.6	17.32	18.45	21.38	32.32	42.73	60.45
synth	0.17	0.75	3.07	6.45	9.3	13.5	33.17	25.93	36.13	79.56
irc	0.1	0.56	2.53	2.83	5.5	7.56	10.04	14.17	20.86	20.94
browser	0.14	0.46	2.64	3.82	7.16	8.88	19.1	14.55	85.74	180.29
interp	0.07	0.33	1.3	2.58	4.96	7.13	11.88	18.44	18.63	23.99
games	0.11	0.44	1.68	3.01	5.23	6.91	8.95	11.77	17.54	16.41
sunit	0.26	1.1	2.26	3.65	6.3	7.24	7.68	10.09	9.24	7.21
pda	0.08	0.76	2.54	5.86	7.65	11.07	16.46	22.16	30.31	38.49
Overall	0.12	0.56	2.27	4.67	8.18	10.6	18.54	20.6	28.53	46.86

Table 6. Speed of the inferencer. Entries gives the average speed in seconds for inferences of instance variables in one component, using the given pruning threshold. The "overall" entries on the last line are averaged across all individual type inferences; thus, they are weighted averages of the component averages, weighted by the number of instance variables within each component.

	50 nodes	150 nodes	500 nodes	1000 nodes	1500 nodes	2000 nodes	2500 nodes	3000 nodes	4000 nodes	5000 nodes
rbparse	33.9	33.9	30.4	32.1	33.9	35.7	39.3	33.9	35.7	39.3
mail	28.6	35.7	31.4	38.6	40.0	38.6	41.4	35.7	38.6	37.1
synth	25.1	27.3	32.8	35.5	36.1	36.1	35.5	35.5	36.6	36.1
irc	37.9	41.4	41.4	50.0	50.0	48.3	51.7	56.9	53.4	62.1
browser	8.7	8.7	13.0	13.0	13.0	8.7	13.0	13.0	13.0	13.0
interp	23.7	24.6	23.7	23.7	24.6	34.8	35.7	34.3	35.3	34.8
games	55.0	64.2	63.3	63.3	63.3	75.0	75.0	74.2	74.2	75.0
sunit	21.4	42.9	35.7	28.6	28.6	42.9	42.9	50.0	50.0	50.0
pda	31.8	34.1	31.8	36.4	36.4	38.6	38.6	38.6	38.6	38.6
Overall	31.0	34.7	34.8	37.0	37.6	42.3	43.3	42.4	43.0	43.7

Table 7. Precision of the inferencer. Entries give the percentage of inferred types considered by a human as "precise" for instance variables in one component using one pruning threshold. As in Table 6, the "overall" entries are averaged across inferences, not averaged across the averages in the table.

For external validation, a complete listing of the type inferred for each variable is available on the net [21], and will be included in the longer report describing this work [22].

The precision varies from 31.0% to 43.7%, depending on the pruning threshold. This level of precision is clearly enough to be useful for many purposes, *e.g.*, program understanding tools which present type information directly to the programmer. The speed varies from 0.12 seconds to 46.86 seconds on average, again depending on the pruning threshold. Again, this level of performance is clearly high enough to be useful in various tools.

The best choice of pruning threshold depends on whether speed or precision is more important. If speed is the most important, then the extreme choice of a mere 50 nodes appears like the best choice. The inferences are very rapid, while precision is a reasonable 31.7%. If precision is more important, then the data suggest that 2000 nodes is a good choice; thresholds above 2000 nodes use much more time while yielding only a small improvement in precision. Of course, the best choice depends on the application; if precision is extremely useful and time is cheap, then higher thresholds would still make sense.

6 Future Work

While **DDP** is effective as it stands, there are several areas of investigation that remain open: extending the algorithm for exhaustive analysis, augmenting the justification rules, and studying pruning techniques.

6.1 Exhaustive Analysis

The approach of this paper allows an analysis of the entire program, but not efficiently: one could execute the algorithm against every variable in the program, one by one. The major inefficiency in this approach is that much information is calculated but then discarded: most queries require a type for multiple variables to be calculated, not just the type of the target variable.

For an efficient exhaustive analysis, it is desirable to keep old results and to reuse them in later queries. However, subgoal pruning adds a complication: distant subgoals of the target goal are more strongly affected by pruning, and thus have relatively low precision. At the extreme, if a subgoal is distant enough that it was in fact pruned, then there is no benefit from reusing it. Thus, it is important to consider how close to the target a goal was before it is reused.

Additionally, it is probably desirable to run multiple queries simultaneously. To choose the queries to run, one could start with an individual query and then promote the first k subgoals created to additional target goals. With this approach, all $k+1$ target goals are likely to contribute to each other and to need similar subgoals. Thus a small increase in the pruning threshold should allow $k+1$ targets to be computed simultaneously.

6.2 Justification Rules

The presence of subgoal pruning allows *speculative* justification tactics to be used, because if they are too expensive, the algorithm will eventually prune them away. Thus, subject to the constraints of soundness, an analysis designer can add more tactics at whim—one tactic for any situation that seems reasonably likely to occur. The algorithm can try the most ambitious available tactic first, and if that is too expensive, it can fall back on other tactics.

One particular way the justification tactics should be augmented is to account for *data polymorphism*. Data polymorphism, in object-oriented languages, is the use of a single class to reference different other classes at different points in a program. In particular, collection classes exhibit data polymorphism. In **DDP**, values retrieved from most collections have type $\top$, because the analysis does not track which additions to collections match which removals from collections. It may be possible to adapt the extension of **CPA** reported by Wang and Smith[23] to work with **DDP**.

A smaller way the justification tactics should probably be augmented is to add a second "senders-of" justification tactic. When finding the type of a parameter, or the flow through a return statement, it is often necessary to find which methods invoke a particular method. Note that while an exhaustive algorithm such as **CPA** may efficiently accumulate call-graph information in the course of analyzing type information, in a demand-driven algorithm the call graph must be queried specifically.

The only senders-of tactic that **DDP** uses is, first, to find all message-send expressions whose selector matches the current method, and then to find and check the type of the receiver of each one. This tactic fails when there are a very large number of `send` statements with a matching selector. An alternative tactic would be to reverse the order of the search: instead of finding all message-send expressions with the correct selector and then filtering those by the receiver type, one could find all variables with a proper receiver type and then filter those by looking at the message selectors sent to that variable. That first step of the query might be called an "inverse type query," because it asks which variables hold a particular type. Such queries would seem straightforward to implement for many classes: start from mentions of the class, then trace flow forward to instantiations, and finally trace flow forward from there. There are complications, such as classes that arise from language constructs other than instantiation, and it does not always work, *e.g.*, if the flow trace leads to $\top_{fp}$, but the general approach seems promising.

6.3 Pruning Algorithm

The pruning algorithm of **DDP** is a not sophisticated. A better algorithm could prefer to prune type goals that only contribute to the call graph; such a pruning has a relatively graceful effect. Further, a better pruner could attempt to make choice prunings that would remove large numbers of subgoals at once. Finally, it may help to prune goals that are already very imprecise. Such goals are unlikely to be useful; additionally, the individual computations on large union types are more expensive than computations on singular $\top$ tokens. The time could better be spent on goals that are more likely to yield a useful result.

6.4 Other Analysis Problems

DDP infers types for Smalltalk, but it could just as well be used for other dataflow problems and other languages. For example **DDP** would likely perform well at escape analysis in Java, or call graph construction in Scheme.

7 Conclusion

This work shows that practical type inference is possible for dynamically typed programs that are hundreds of thousands of lines long. The path to scalability relies upon *demand-driven analysis* to focus the search, and *subgoal pruning* to manage resources. **DDP** is a straightforward instantiation of this approach that works.

However, we'd like to conclude with a message larger than the specifics of **DDP**: the potential of applying AI-based, resource-bounded search techniques to intractable program-analysis problems. The key attraction of these techniques is their ability to match the computational resources available for a task to the specific complexity of different instances of that task. We applied this notion to the task of reasoning about Smalltalk programs because we are personally enthusiastic about this style of language and programming. But it may well be applicable to other programming languages and other analytic tasks, as well. Thus the manner in which we solved our problem, and the general outline of its solution, is quite likely to be of broader interest than the particulars of our intended application.

8 Acknowledgements

We are grateful to our anonymous reviewers for comments that have contributed to the clarity of the paper.

References

1. American National Standards Institute: ANSI NCITS 319-1998: Information Technology — Programming Languages — Smalltalk. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA (1998)
2. Kay, A.C.: The early history of smalltalk. In: The second ACM SIGPLAN conference on History of programming languages, ACM Press (1993) 69–95
3. Kogge, P.M.: The Architecture of Symbolic Computers. McGraw-Hill (1991)
4. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)

5. Grove, D., Defouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA). (1997)
6. Shivers, O.: The semantics of scheme control-flow analysis. In: Partial Evaluation and Semantic-Based Program Manipulation. (1991) 190–198
7. Suzuki, N.: Inferring types in smalltalk. In: Conference record of the 8th ACM Symposium on Principles of Programming Languages (POPL). (1981) 187–199
8. Barnard, A.J.: From types to dataflow: code analysis for an OO language. PhD thesis, Manchester University (1993)
9. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Proc. of ECOOP. (1995)
10. Flanagan, C., Felleisen, M.: Componential set-based analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **21** (1999) 370–416
11. Garau, F.: Inferencia de tipos concretos en squeak. Master's thesis, Universidad de Buenos Aires (2001)
12. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University (1991)
13. Heintze, N., McAllester, D.A.: On the cubic bottleneck in subtyping and flow analysis. In: Logic in Computer Science. (1997) 342–351
14. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. ACM SIGPLAN Notices **35** (2000) 281–293
15. Reps, T.W.: Demand interprocedural program analysis using logic databases. In: Workshop on Programming with Logic Databases (Book), ILPS. (1993) 163–196
16. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: Symposium on Principles of Programming Languages. (1995) 37–48
17. Agrawal, G.: Simultaneous demand-driven data-flow and call graph analysis. In: ICSM. (1999) 453–462
18. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: SIGPLAN Conference on Programming Language Design and Implementation. (2001) 24–34
19. Dubé, D., Feeley, M.: A demand-driven adaptive type analysis. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ACM Press (2002) 84–97
20. Rich, E.: Artificial Intelligence. McGraw-Hill Book Company (1983)
21. Spoon, S.A., Shivers, O.: Classification of inferred types in ddp experiment. —http://www.cc.gatech.edu/˜lex/ti— (2003)
22. Spoon, S.A.: Subgoal Pruning in Demand-Driven Analysis of a Dynamically Typed Object-Oriented Language. PhD thesis, Georgia Institute of Technology (forthcoming)
23. Wang, T., Smith, S.F.: Precise constraint-based type inference for Java. Lecture Notes in Computer Science **2072** (2001) 99–117