Dynamic Data Polyvariance
Using Source-Tagged Classes

S. Alexander Spoon
lex@lexspoon.org

Olin Shivers
shivers@cc.gatech.edu

Abstract

The **DDP** (Demand-driven/Pruning) analysis algorithm allows us to perform data-flow analyses of programming languages that are dynamically typed and have higher-order control flow, such as Smalltalk or Scheme. Because it is demand-driven and employs search pruning, it scales to large code bases. However, versions of the algorithm previously described [19] do not handle *data polymorphism* well, conservatively merging separate data flows that go through distinct instantiations of a collection type. In this paper, we describe a new extension to **DDP** that helps to disentangle these flows, permitting more precise results. The extension is based on *source-tagging* classes so that each reference to a class in the source code yields a subdivision of the type associated with that class. An initial implementation of this polyvariant analysis has been added to the **DDP**-based tool Chuck, a part of the integrated Squeak program-development environment; we show examples of the tool in action.

1 Data polymorphism and analysis precision

In his dissertation, Agesen distinguished two kinds of polymorphism that occurred in object-oriented programs [3]. *Parametric polymorphism* arises when different classes provide different code to handle the same message. *Data polymorphism* (in Agesen's words) describes "the ability of a slot (instance variable) to hold objects with multiple object types." Generic "container" or "collection" classes such as lists, tables and arrays are the standard example of data polymorphism: the one `Vector` class can be used to create both an integer vector and a string vector.[1] As Agesen pointed out, data polymorphism can induce significant loss of pre-

---

[1]This terminology may be slightly confusing to programmers from the Hindley-Milner type-inference camp, who would more likely use the term "parametric polymorphism" to describe what Agesen calls data polymorphism. We hew to Agesen's terminology, since the analyses we're describing in this paper primarily concern object-oriented languages, and are related to Agesen's work.

cision in analyses that perform, or are dependent on, type inference. A data-flow analysis in this setting will typically merge the types of all the values that flow into distinct instances of some collection class. From the analyzer's point of view, if an object flows into *one* instance of a collection class, it will then flow out of *every* instance of that collection class. So, for example, if the program has two completely distinct vectors, one containing integers and the other containing strings, analysis will show that a single fetch from either of these vectors could produce either an integer or a string. The loss of precision can hurt important analyses for dynamic languages, such as control-flow analysis [16, 17], type inference [3] and dead-code removal [3, 21].

To address this problem, one can enrich the analyzer's type system to partition objects more finely than by class. Instead of all instances of class `Vector` being in the same type, those instances can be subdivided into the types (Vector, l_1), (Vector, l_2), ..., (Vector, l_n) for some sequence of discriminators $l_1 \ldots l_n$. The difference is shown in Figure 1 and Figure 2. This partitioning segregates flow paths that go through the class: flow into any object of type (Vector, l_i) can only flow out of an object of that same type.

The choice of partitioning matters. A good partition leads to a flow graph like that in Figure 2, whereas a poor one leads to a flow graph like that in Figure 3.

For languages constrained by generous amounts of static semantics (such as Java [7]), an effective partitioning strategy is that of Wang and Smith's **DCPA** algorithm: subdivide objects according to which `new` expression instantiated them [23]. This approach yields a true partition because every object must have been instantiated by exactly one `new` expression; objects created on line 134 must be different from objects created on line 431.

For extremely dynamic languages such as Smalltalk, however, this approach is ineffective by itself. The problem is that, in Smalltalk, object creation is not a primitive syntactic form. It is a single primitive method (called `basicNew`) that is triggered indirectly by various instance-creation methods around the program.[2] Smalltalk classes are themselves objects, and this is exploited in the object-creation protocol, with the classes being passed through the creation chain to the actual `basicNew` step.

This paper describes a partitioning strategy that is effective in the presence of the dynamic and reflective facilities of a language such as Smalltalk. The approach is based on *tagging* class objects with context information taken from abstract semantics that can be used to segregate instances of the class.

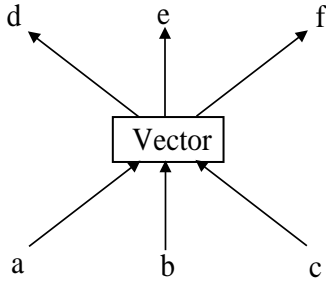[2]We are simplifying slightly for clearer exposition.

Figure 1. Without data-polyvariant analysis, a type inferencer intermixes all flows that go through a class's instance variables. For example, according to this analysis-level flow graph, objects in flow position `a` might flow to any of `d`, `e`, or `f`.
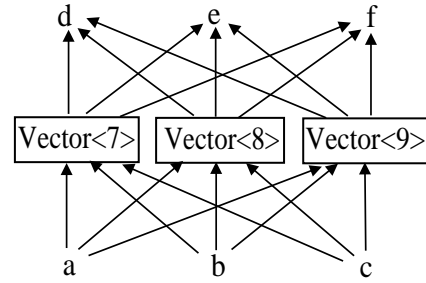

Figure 3. This poor choice of partitioning leads to a more expensive analysis with no improvement in precision. As in Figure 1, the analyzer will predict that flow from any of `a`, `b`, or `c` can reach any of `d`, `e`, or `f`.
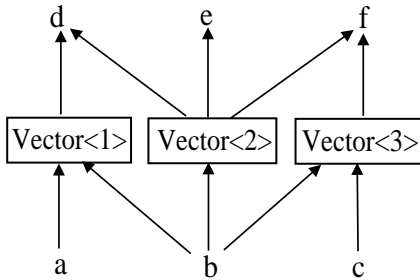

Figure 2. If instances of a class can be partitioned, then a type inferencer can, to an extent, segregate flows through the class's instance variables. For example, objects in flow position `a` can now be seen to flow to `d` but *not* to positions `e` or `f`. On the other hand, flow from `b` still reaches three positions. This partitioning improved the flow analysis of `a` and `c` but not of `b`.

Figure 4. The semantics of Smalltalk are that classes are shared and singular. All instances of class `Point` (`3@4`, `6@4`, etc.) have a reference to the same class object.

2 Source-tagged classes

Source-tagged classes give a way to approximate the partitioning approach of the proven **DCPA** algorithm, even though Smalltalk only has one `basicNew` method instead of Java's many separate `new` expressions throughout the program. The approach exploits the common idiom that most objects are created with a message-send expression whose target is the immutable global variable that is the primary reference to the class object. Common examples are "`ValueHolder new`" and "`Point x: 3 y: 5`." In this idiom, the constructor method (`new` and `x:y:` in these two examples, respectively) invokes the `basicNew` method on the class to instantiate the class and then invokes a sequence of methods on the resulting object to initialize that object with the specified parameters.

The partitioning approach of this paper, then, is to attach a *source tag* to all distinct references to classes in the source. Figure 4 and Figure 5 depict the general idea. Each location in the program text where the class is mentioned has its own source tag. The abstract semantics associated with the type inference evaluates such a class reference to its tagged value. The tag is preserved as the abstract value flows through the program during the analysis' abstract interpretation. When an object is instantiated by sending the primitive `basicNew` method to the class object, the tag is transferred to the abstract object thus created. Thus we get the effect of **DCPA** in this more dynamic setting: two different occurrences of "`ValueHolder new`" in the source code will cause two distinct abstract values to be created by the analysis. Hence, when an abstract value later flows into one of these two instances, it won't erroneously "tunnel" over
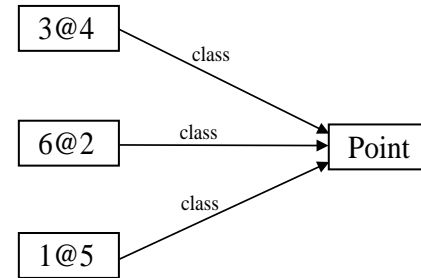
to the other one.

Note that we are assuming enough abstract-context information is present in the basic analysis to distinguish message sends that occur in type-distinct contexts (that is, the kind of distinction we already get from Agesen's CPA analysis). Our concern here is only to distinguish distinct instantiations of a class that have the same type in the base analysis.

3 DDP

The source-tagged technique we present in this paper is an extension to a "base" algorithm that is capable of basic data-flow analysis in the presence of the dynamic-language features we wish to support. The algorithm we are using for our base is **DDP** [19], a new analysis framework specifically designed for performing data-flow analyses on higher-order, dynamically typed languages, such as Smalltalk [4, 11] or Scheme [1]. Besides its ability to handle the analytic challenges of these languages, **DDP**'s other strength is that it scales well to large code bases. For example, the Squeak [10] open-source implementation of Smalltalk now includes a **DDP**-based tool, Chuck, to provide type inference and semantic-navigation services to the Squeak program-development environment. Chuck provides interactive performance, replying to individual queries over the entire 300,000-line Squeak code base in under five seconds; this level of service is supported when queries are interleaved with changes to the code base [20].

DDP provides this kind of performance by means of two key ideas taken from technologies developed by the AI community, which has long been accustomed to performing heuristic searches in in-
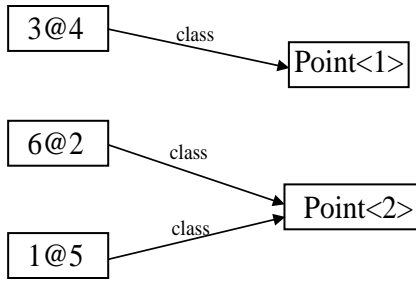
Figure 5. For DDP/CT, we can imagine that there are multiple copies of each class, one for each location the class is mentioned in the source code. Each copy is indistinguishable to regular program code—even the equivalence operator (==) does not distinguish them—and thus the compiler does not need to explicitly represent the tags. The tags are present in the abstract semantics, however, and can be reasoned about by a type inferencer.
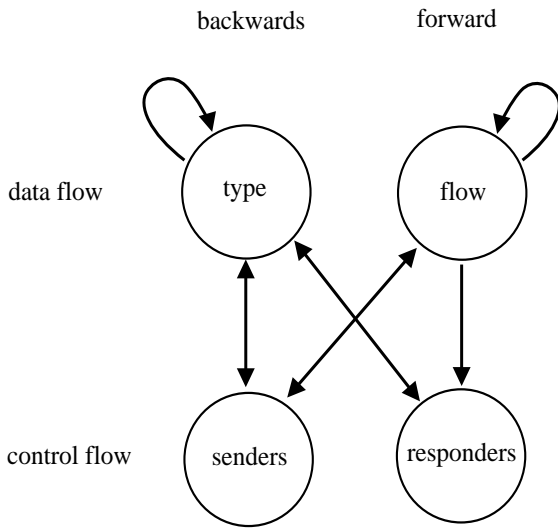

Figure 6. The four queries DDP/CT can answer along with their dependencies on each other.

tractable spaces. First, **DDP** does not analyze the entire program in one invocation. Rather, it is a *demand-driven* analysis that does goal-directed backwards search to satisfy a specific request for information. Thus, a particular analysis will typically traverse only a very sparse fragment of the entire code base, allowing for sub-linear run times.

Second, **DDP** copes with searches that become intractable by *pruning* the goal tree. Subqueries occurring far from the root query are trimmed by providing them with answers approximate enough to be cheaply computable with no further subgoal recursion. The heuristic exploited here is that imprecision can be more easily tolerated far from the root goal than close to it, analogously to the way that chess-playing algorithms use expensive, exhaustive searches early in the search tree but cheap, approximate evaluations deep into the search tree.

3.1 DDP goals

DDP uses four kinds of goals, or queries, to construct a goal tree satisfying an initial root query.

1. A *flow query* asks where the value of a computation could flow.

2. A *type query* asks what kinds of values could flow to a given expression.

3. A *responders query* asks where control could go at a given method invocation.

4. A *senders query* asks which program points could transfer control to a given class method.

These four queries directly correspond to the cross product of {backward,forward} with {data,control}-flow, as shown in Figure 6.

Subgoal recursion occurs when the means of answering a particular goal requires answering one or more subgoals. The arrows in Figure 6 show dependencies between goals and their subgoals. For example:

- A flow query for the argument in a message-send expression depends on a responders query in order to find the methods to which the argument could flow. Thus, there is an arrow in the diagram from flow queries to responders queries.

- A type query for a message-send expression depends on a responders query in order to find what methods might respond and thus contribute a type to the message-send expression.

- A responders query depends on a type query in order to determine the type of the receiver of the message send, which in turn is needed to predict which methods might respond to the message send.

- A senders query depends on type queries in order to filter candidate message-send expressions by the type of the receiver.

Note that most arrows in the diagram go from a control-flow query to a data-flow query or vice versa. Control- and data-flow are tightly interwoven in higher-order, dynamic programming languages [17].

Flow queries

A *flow query* asks where the value produced by some variable or expression will flow when code in the code base runs. This information is useful to see where something is ultimately used. For example, a user can select the variable `RadiansPerDegree` in class `Float`, ask where the value in the variable flows, and be told that it flows to the methods `radiansToDegrees`, `degreesToRadians`, `*`, and `/` in class `Float`. By reading the code of these methods, the user can see that `radiansToDegrees` and `degreesToRadians` refer to the variable in order to convert angles between radians and degrees, and in turn those two methods pass the number to the `*` and `/` methods.

The answer to a flow query is a set of *flow positions*. The following flow positions are possible:

- **Variables**
  For example, `Display` is a flow position designating the values assigned to the `Display` global variable during program execution.

- **Expressions**
  Any expression is a flow position designating the values the expression might produce at run time.

- **Methods**
  For example, method `next` of class `Random` is a flow position designating values held by the receiver (`self`) of the specified method.

These flow positions are additionally discriminated by static contexts as described below in subsection 3.2.

For efficiency reasons, the value *Anywhere* holding all possible flow positions is implemented as a special case requiring only constant space to represent and constant time to process.

Type queries

A *type query* asks what kind of values a variable or expression will hold when the code base runs. The answer to a type query is a set of "concrete types" from the program:

- **Individual class**
  For example, `PlayingCardDeck` is a valid type which includes all instances of that class.

- **Individual symbol**
  For example, `#straight` is a type which includes only the symbol object named `straight`. Tracking symbols is important (1) to handle their use to represent *ad hoc* enumerated types and (2) to handle Smalltalk's `perform:` dynamic-dispatch facility, which is a key control-flow idiom used in real Smalltalk programs.

- **Individual block**
  A block type specifies a particular block from the source code. Smalltalk blocks are akin to Scheme lambda expressions. A block type includes all closure objects which were created by evaluating the specified block. Note that handling blocks with precision is critical for analysis of Smalltalk programs. Smalltalk blocks are first-class values used in a pervasive and fine-grained way. For example, the basic if/then/else construct in Smalltalk is provided by sending two block objects to the boolean selector. Failure to handle blocks in a polyvariant manner would confuse together the control flow of every conditional branch in the entire code base.

Like flow positions, type queries are additionally discriminated by abstract-contour context, which is particularly important for higher-order values such as blocks.

For efficiency reasons, the top type *Anything* holding all possible objects is implemented as a special case, requiring only constant space to represent and constant time to process.

Responders queries

A *responders query* asks what methods or blocks might respond when a particular message-send expression executes. As an extreme example, if one browses to class `BasicLintRuleTest`'s `new` method in the Squeak code base, and selects the message send of `initialize`, the standard syntax-directed query shows 756 potential responders. The Chuck tool uses **DDP** to answer the responders query and shows only one.

Senders queries

A *senders query* asks what expressions might invoke a specified method or block. To find the senders of a method, the analyzer must determine not only that the message sent by the expression is the right name, but that the object that is the target of the expression's send is the right class. For example, performing a senders query on an HTML class's `parse` method should *not* include expressions in the code base that send the `parse` message to VRML or email objects. Type information is a powerful discriminator here—methods with common names such as `initialize` often have only one sender according to **DDP**, but hundreds of senders according to simple syntactic criteria.

3.2 Context

Like many program-analysis algorithms, **DDP** can analyze the same syntactic element (expression, variable, method, ...) under multiple *contexts* or *abstract contours*. This general approach is widely used in program analysis [12]. A context, in general, can be viewed as a predicate on the execution states associated with a control point. Contexts allow an analysis to keep distinct facts about program execution states, and their consequences, that may share a common control point.

One widely studied kind of context is the *call chain* [15]. A call chain specifies the dynamically nested sequence of procedure calls or message sends that are pending at the given execution state. For example, "the immediate caller is statement 3 of method `foo`," or "the immediate caller is statement 3 of method `foo`, and its caller is statement 4 of method `bar`." The length of the call-chain segment to which the analysis is sensitive is typically limited by a constant that is a parameter of the analysis algorithm. The number of contexts per control point can be exponential in the length of the call chains, with an exponent base that is linear in the size of the program. Two of the many algorithms that use call chains are **k-CFA** [18] and Emami's points-to analysis [5].

The kind of context used by **DDP**, however, might be called *parameter-types context*. A parameter-types context specifies the types of method parameters in the current lexical scope, e.g., "the first parameter is an `Integer` and the second is a `Float`." In an object-oriented language, a parameter-types context can also specify the type of the method receiver, e.g., "the receiver is an `Integer` and the first parameter is a `Float`." In a language with blocks (or lambda expressions), a parameter-types context is more general and can additionally include types for a chain of lexically nested blocks within a method.

The key to this kind of contour abstraction is that, in object-oriented and dynamically-typed languages, control flow (e.g., method dispatch) fundamentally depends upon this kind of type or class information. Type-based contour abstractions provide precisely the kind of discrimination that is needed to analyze the basic flow behaviour of a program.

There are subdivisions within the general approach of parameter-types contexts. The Cartesian Products Algorithm (**CPA**) uses contexts where each parameter type is a specific class; thus, the contexts for each method correspond to the cartesian product of the classes in the type of each parameter [2]. To contrast, the Simple Class Sets (**SCS**) algorithm chooses one parameter-types context for each combination of types that appear at some call site in the program [8]. **DDP** uses **CPA**-style parameter-type contexts. As a

result, it is sometimes necessary to subdivide a context into multiple partitions in order to fit the constraint of **CPA** and then analyze separately under each partition.

Context is interwoven throughout the analysis. Almost every place that a syntactic element appears, it is adjoined to an abstract context:

- *Flow positions* are specified not only as a variable, expression, or (in the case of self-of-method positions) a method, but also with context. For example, one possible flow position would be "variable x of method `foo:`, under a context where `foo:`'s parameter is a `SmallInteger`." The presence of context in flow positions means that flow queries can produce more specific responses than they otherwise could. Instead of simply describing the variables through which a value can flow, they can describe the types of objects that will be present in the environments (lexical scopes) surrounding those variables.

- *Block types* mention not only a block, but also a context in which the block expression was evaluated. This context can specify the types of any parameters that are in scope of the block. Later, when the block's contents are analyzed, the analysis can be improved by using the recorded types of those parameters. As with flow positions, having a context associated with a block type allows the answer to a type query to include not only the blocks to which a variable might refer, but also the types of objects in the environment around that block at the time the block was created.

- *Type queries* can ask about a variable in context instead of just a variable. For example, a type query can ask, "what is the type of x under a context where the first parameter of its lexically containing method is a `SmallInteger`?" This is a critical form of analysis polyvariance for polymorphic code.

- *Responders queries* use context both for the queries and the responses. The queries can include a context along with the message-send expression. The responses include not only a set of methods and blocks that can respond to the message send, but also the contexts under which they might respond. As an example, the query "who responds to x + y, in a context where x is a `SmallInteger`?" could have as an answer, "`+` in class `SmallInteger`, where both the receiver and the first argument are `SmallInteger`'s."

- *Senders queries*, likewise, can return a set of expressions that can invoke a method or block along with the context where those expressions might invoke the method or block.

As a minor technical note, not all contexts can be applied to all variables, expressions, or flow positions—a context may only specify types for parameters that are in the scope of the associated syntactic item. As a result, it is sometimes necessary to *broaden* a context before it can be applied to one of these items. Frequently when we write that an item should be considered in some context *ctx*, we really mean that it should be considered in context *ctx′* where *ctx′* is a broadening of *ctx* to be sensible for the item under discussion.

4 Standard solution strategies

This section describes the previously unpublished solution strategies that the base **DDP** uses to solve the above kinds of goals. These strategies follow straightforwardly once the goals and their answers have been formulated. The strategies are listed below in order to show the structure and interplay of the base analysis' deductions, which will set the stage for developing the new extensions described in the following section.

While reading these solution strategies, be aware that goals are solved under the assumption that their subgoals have correct tentative solutions. This assumption is usually false the first time a goal is updated, because all of the goal's subgoals are freshly created and almost certainly have over-specific tentative solutions. As the analysis progresses, such a goal will be revisited when its subgoals have more reasonable tentative solutions. The algorithm does not terminate until the entire goal network has stabilised into a consistent solution to the top-level, root query. Thus, it is easiest on a first reading of these solution strategies to assume that all subgoals requested have complete and correct answers—just as the analyzer does.

4.1 Responders queries

A responders query is of the form, "what methods or blocks reply to the message-send expression '*rcvr sel arg0 … argn*' if the expression is executed in context *ctx*?" This expression requests that the method named *sel* of the object *rcvr* be invoked, and that the method be passed *arg0…argn* as arguments.

To answer a responders query, **DDP** begins by posting type queries for *rcvr* and for each of the arguments *arg0*, …, *argn*. The solution to the type query on *rcvr* is used to determine which methods and blocks respond, while the argument types are used to determine the context under which those methods and blocks will execute.

To find the responding methods, the analyzer considers every class that is a member of the receiver's type and finds the method that will respond if message *sel* is sent to an instance of the class. If the responding method happens to be one of the primitive block-evaluation methods (`value`, `value:`, etc.), then the analyzer examines the receiver type to determine what blocks are included in the receiver and thus what blocks can respond.

To find the possible responding contexts, **DDP** begins by taking the cartesian product of the receiver type and all of the argument types, just as **CPA** does. For methods that respond to the message-send, these cartesian products can be used directly, and the algorithm returns the cartesian product of the responding methods and each responding context.

For blocks that respond, the contexts cannot be used directly. A further step is required to create the responding contexts. Each context from the cartesian product of the receiver type argument types is combined with context associated with each block type in the receiver type. The context from the cartesian product supplies the types of the block's own parameters, while the types in the block type's associated context supply the type of the receiver and the types of parameters that are lexically visible from within the block.

4.2 Senders queries

A senders query asks, "what message-send expressions invoke the block or method *methblk*?" The answer to this question includes three kinds of sending expressions: regular message sends, `perform:` message sends, and block evaluations.

Regular message sends are the most straightforward of the three to find. If *methblk* is a block instead of a method, then there are no regular message sends that invoke it. Otherwise, **DDP** begins by finding all message-send expressions whose selector matches the method's selector. For each such expression, it posts as a subgoal a type query that attempts to find the type of the receiver. If sending

the method's selector to objects in that type could possibly invoke *methblk*, then that message-send expression is considered a possible sender of *methblk*.

Smalltalk methods can also be invoked indirectly via the `perform:` family of methods. When the base `Object` class's `perform:` method is invoked at run time, the first argument to the method must be a symbol object that names another method to execute. The system then sends the message named by the symbol to the current receiver with the following arguments used as the parameters. **DDP** makes a simplifying assumption in order to make senders-of goals tractable in the face of the `perform:` method. **DDP** assumes that at the normal application run time, all symbols passed into such a method appear somewhere in the program text. That is, DDP does not consider control flow that might arise from the program computing a string, converting the string to a symbol, and then invoking a method by means of an indirect message send via `perform:` with the computed symbol. This restriction renders the analysis feasible, but still captures the idiomatic uses of `perform:` pervasive in certain kinds of Smalltalk code (e.g., `perform:` dispatch is critical in the construction of GUI event loops in Smalltalk applications).

Proceeding from this assumption, **DDP** finds `perform:` senders by tracing flow *forward* from each occurrence of the method's name as a symbol literal; this is done by recursively posting a flow query for each such literal. Most method names do not appear as a symbol anywhere in the program, in which case there are no `perform:` senders of that method. For method names that do appear as symbol literals, the analyzer sifts through the responses to the flow query, finding all message-send expressions where the first argument could be the symbol; it then uses a type query to check whether that message-send expression might invoke a `perform:` method. If it might, then that message-send expression is considered a possible invoker of *blkmeth*.

Finally, for the third kind of message send, if *blkmeth* is a block, then it is invoked via a block-evaluation primitive method (`value`, etc.). To find such senders, **DDP** traces the flow of *blkmeth* using flow queries. At each message-send expression where the object flows to the receiver, the analyzer checks where a block evaluation method might be invoked by that message-send expression. If so, then the message-send is considered a potential sender of *blkmeth*.

4.3 Type queries

A type query is of the form, "to what type of objects does *varexp* evaluate in context *ctx*?" To answer this question, **DDP** considers six kinds of syntax: literals, references to classes, `self`, assignment statements, parameters, and message-send expressions.

If *varexp* is a literal then the type inferred for *varexp* is simply the type of the literal. If the literal is a small integer, then the inferred type is $\{\!|\texttt{SmallInteger}|\!\}$; if it is a string, the inferred type is $\{\!|\texttt{String}|\!\}$; and so on.

If *varexp* is a reference to a class, then the inferred type is the metaclass for that class. For example, if *varexp* is `Array`, then the inferred type is $\{\!|mclass(\texttt{Array})|\!\}$, where $mclass(\texttt{Array})$, often written `Array class`, is the class of `Array` itself.

If *varexp* is `self`, a reference to the current receiver, then **DDP** uses one of two simple strategies. If *ctx* specifies a type other than *Anything* for the current receiver, then that type is inferred as the type of *varexp*. Otherwise, the type inferred for *varexp* is the union of

the receiving method's class along with all of its direct and indirect subclasses.

If *varexp* is a variable that is modified by assignment statements, then **DDP** posts as subgoals a type query for each right-hand side that is assigned to *varexp*. The context used for each type-query subgoal is *ctx* itself. The type of *varexp* is inferred to be the union of the types of the right-hand sides.

If *varexp* is a parameter, then it is not modified by assignment statements. Instead, it takes on values by message-send expressions: message sends provide arguments that are bound to the parameters of the responding method. If *varexp* has a type specified in *ctx*, then, as with the similar case for `self`, **DDP** simply uses the type specified by *ctx*. Otherwise, **DDP** posts as a subgoal a senders query to determine what expressions invoke the method or block for which *varexp* is a parameter. Once those expressions are located, **DDP** posts a type query for the actual arguments that correspond to *varexp*—e.g., if *varexp* is the third parameter of its binding method, then the corresponding actual argument would be the third argument for regular senders and the fourth argument for `perform:` senders. The type inferred for *varexp* is then the union of the types of all of the corresponding actual arguments.

Finally, if *varexp* is a message-send expression, then **DDP** needs to find the methods or blocks that respond to the message send. Thus **DDP** begins solving the goal, in this case, by posting a responders goal on *varexp*. The responders goal returns a number of methods and blocks, each paired with a context. For each block/context pair, **DDP** issues a type query on the last expression in the block—that is, the expression providing the block's return value. Similarly, for each method/context pair, **DDP** scans the method to locate the method's return statements, and issues a type query for each return expression. The inferred type for *varexp* is the union of the solutions to all of these type queries.

4.4 Flow queries

A flow query is of the form, "where do values flow, starting from *fpos*?" **DDP** solves these queries by reducing them to *one-step flow queries* of the form, "where can values flow from *fpos*, in a single step of execution?" To answer a normal flow query, **DDP** begins by posting a one-step flow query for the initial position. For each new flow position that is part of the one-step flow query's solution, **DDP** posts another one-step flow query. For each flow position in the solutions to these queries, **DDP** posts yet another query, and so on, until none of the flow positions flows to a new location. The solution to the original flow query is then the one-step closure: the union of *fpos* with the solutions to all of the one-step flow queries.

If *fpos* is a flow position for a variable, then a one-step flow query on *fpos* simply returns all expressions that directly reference the variable. If *fpos* is a flow position for a method, then the solution is similarly simple: the one-step flow is inferred to be all `self` expressions within *fpos*'s method.

A one-step flow query for an expression *exp* is more complicated to answer. Its solution must account for assignment statements, returns from methods and blocks, and message-send statements.

If *exp* is the right-hand side of an assignment statement, then the one-step flow from *exp* is the variable on the left-hand side of the statement with the same context as *fpos*.

If *exp* is immediately returned from a method or block, then the value *exp* produces at run time will flow out of the method or block and into the message-send expression that invoked it. To find the one-step flow in this case, **DDP** posts a senders query on the block or method. The one-step flow from *fpos* is precisely the answer to this query.

If *exp* is the receiver term of a message-send expression, then the value produced by *exp* at run time will become the receiver (`self`) of whichever method responds to the message send. To find the one-step flow in this case, **DDP** posts as a subgoal a responders goal on *exp* in order to find any methods that can respond—blocks that respond are ignored, because there is no equivalent to `self` expressions for accessing the currently executing block. The inferred one-step flow for *fpos* includes each method/context pair that this responders goal returns.

Finally, if *exp* is an argument to a message-send, then the value produced by *exp* at run time will become a parameter to the responding method. In this case, **DDP** again posts a responders goal for *exp* under the context that is part of original *fpos* query. This time, however, it does not ignore blocks that are included in the responder. The one-step flow of *fpos* includes the appropriate parameter of each block or method that responds.

5 DDP/CT

Our class-tagging extension to **DDP**, called **DDP/CT**, extends the base analysis in five ways:

1. We extend the type system to allow class types to be subdivided using *source tags*.

2. We add a new kind of goal, the *inverse type goal*.

3. We add a second *subgoaling schema* or solution strategy for answering senders goals that uses inverse type goals.

4. We add a new kind of goal for finding the *type of array elements*.

5. We *augment flow goals* so that they can trace the flow of just those objects within a specified type.

Source tags are the core of **DDP/CT**'s extensions. They provide a mechanism for subdividing the set of objects that are instances of one class, thereby providing a way to segregate data-flow paths through such objects.

The other four extensions are needed to exploit this new subdivision. The new strategy for senders goals is needed to trace backwards from a class's initialization methods to callers that feasibly invoke the methods for a particular partition of the class's instances; with the standard **DDP** strategy, all invokers of the initialization methods would be considered feasible, leading to the intermixing that subdividing the types was intended to prevent. The new inverse type goals, in turn, are required to support this new strategy for answering senders goals.

The array-element type goals are added because arrays are widely used data-polymorphic objects in Smalltalk, not only as data-structures in their own right, but also as the underlying storage for many other collection classes, such as hash tables. We hope that source-tagged types will finally provide a way to analyze these uses precisely. Type-specific flow goals have been added as a simple way to improve the precision of flow goals by avoiding flow paths of objects other than the interesting ones.

5.1 Source-tagged class types

DDP/CT adds a *source tag* to class types, as described earlier in section 2. A class type $\{\!\{C\}\!\}$ in **DDP** includes all objects that are instances of class C. In **DDP/CT**, a class type can also include a *source tag* l and be of the form $\{\!\{C, l\}\!\}$. The type $\{\!\{C, l\}\!\}$ includes precisely those instances of C that, in the abstract semantics, are tagged with source tag l.

Tagged types are introduced to a running type inference whenever there is a type goal for an expression that simply reads the primary global variable holding a class. (Recall that in Smalltalk, classes are themselves objects, not simply the compile-time constructs found in less dynamic languages.) Instead of answering such a goal with a simple class type $\{\!\{C\}\!\}$ as the base **DDP** would, **DDP/CT** answers the goal with some tagged class type $\{\!\{C, l\}\!\}$. Source-tagged types are then automatically propagated through the goal network, allowing other goals to take advantage of the more specific types to generate more specific inferences.

5.2 Inverse type goals

An *inverse type goal* requests a flow position that includes all program locations that could produce an object of a specified type. It is an inverse of a type goal: instead of asking what type is held by a specified flow position, it asks what flow positions hold a specified type. Inverse type goals are used to support the improved senders-of goal described below. Inverse type goals, of all the goals used by **DDP/CT**, take particular advantage of source tags.

The specified type must be a source-tagged class type. To solve an inverse type goal for source-tagged type $\{\!\{C, l\}\!\}$, **DDP/CT** uses one of two strategies depending on whether C is a regular class, e.g. `Point`, or a metaclass, e.g. `Point class`, of which `Point` itself is an instance.

If C is a regular class, then $\{\!\{C, l\}\!\}$ includes objects that were created by the instantiation method: that is, the primitive method `basicNew` in the Squeak dialect of Smalltalk. To solve such a goal, **DDP/CT** simply traces the forward flow (by posting flow goals) of the return value from the `basicNew` method[3] under an assumed context that the receiver is of type $\{\!\{mclass(C), l\}\!\}$. We use "*mclass(C)*" to mean the metaclass of class C. Solving this goal will require finding the precise senders of the `basicNew` message under these assumptions as described in the next section.

If C is a metaclass, then $\{\!\{C, l\}\!\}$ includes the class *rclass(C)* with tag l, where we use *rclass(C)* to mean the regular class whose metaclass is C. Aside from direct data flow, such an object can only enter the computation from two sources: the program executes the expression with tag l, or the program invokes the reflective `class` method on an instance of $\{\!\{rclass(C), l\}\!\}$. The `class` method returns the class of the receiver of the message, and it is frequently used in idiomatic Smalltalk. For example, it is used (indirectly) by the `copy` method of the `Collection` class in order to create a new collection of the same class as the receiver. Therefore, if C is a metaclass, **DDP/CT** traces flow forward from two places: the expression with tag l, and the `class` method under a context where the receiver type is $\{\!\{rclass(C), l\}\!\}$.[4]

---

[3]There are actually a small number of such methods in Squeak, and the analyzer must trace all of them.

[4]We freely admit that the complexities of handling Smalltalk's more extreme reflective features have caused us frequently to con-

Some exceptions should be noted. A fixed set of primitive Smalltalk classes have special syntax for creating instances of that class; these classes are not typically instantiated by means of sending `new`-style creation messages to the class. Examples are blocks, which have their own syntax, and numbers, which can appear as literals. **DDP/CT** reverts to special-case handling to implement inverse type queries on these classes in a straightforward way.

5.3   Senders goals

Recall from subsection 3.1 that a *senders goal* in **DDP** finds those expressions in the program that can invoke a specified method in a specified context. The strategy **DDP** uses to find those senders is: first, find all message-send expressions that invoke a method of the appropriate name, and second, check that the type of the receiver (which must be inferred using a subgoal) is consistent with the expression invoking the method.

A potential difficulty of this approach arises if there are a large number of message-send expressions whose message name matches the name of the queried method. For example, when trying to find the senders of the `AtomMorph` class's `initialize` method, the standard strategy would consider hundreds of potential message-send expressions, generate a type query for each one of them, and, most likely, both generate a large number of subgoals and include a large number of false positives. Worse, consider querying for the senders of method `at:put:` in class `Array`, perhaps as part of an effort to find the type of elements that could be added to a particular set of interesting arrays. In the standard Squeak code base, there are over one thousand senders of `at:put:` to sort through, and many of them do, in fact, invoke `Array`'s `at:put:` method. Potentially only a small number of them invoke `at:put:` on the particular array *objects* that are of true interest, but if the question is formulated as "who invokes `at:put:` in `Array`," then the answer to the question is forced to include a large number of extra senders in order to be correct.

DDP/CT therefore uses an alternative strategy if the specified context includes a non-trivial receiver type (i.e., not the top type *Anything*). If the receiver type of the method is specified, then the method in that context can only be invoked by a message-send expression where the receiver is in the specified type. The alternative senders-goal strategy uses this fact. It has as a subgoal an inverse type goal for the specified receiver type. The answer to this subgoal includes all expressions in the program that can hold an object of the specified type. The alternative strategy then selects as possible senders those message-send expressions whose receiver is in the inverse type goal's answer and whose message selector matches the method being queried.

In other words, the alternate strategy swaps the roles of the two selection criteria. Instead of applying a semantic filter to the results of a base syntactic query, it syntactically filters the results of a semantic query.

---

sider the benefits of a more strictly phased language design. The lure of adapting our analyses to work with full-fledged Smalltalk is the payoff of having 300,000 lines of source code and a large community of coders to exercise our technologies. The test bed thus provided will provide the experience that will, we hope, direct a following round of language design. The challenge is to design annotations that provide leverage for the static analyses that drive programmer-support tools, without compromising the flexibility and power we expect of dynamic languages.

5.4   Array-element type goals

Smalltalk arrays are treated as regular objects. There is no special syntax for accessing them. Instead, an array is an object *a* that handles operation "*a* `at:` *i*" to retrieve the element at index *i*, and "*a* `at:` *i* `put:` *e*" for storing element *e* into the array at index *i*. Other objects in the system respond to the `at:` and `at:put:` messages, doing non-array operations in response to them, and thus an expression such as "`a := b at: i`" might or might not perform an array operation. In fact, different executions of this same statement might sometimes invoke an array operation and other times not, depending on the class of object to which `b` is bound at each execution.

The type goals of **DDP** find a type for a variable, but Smalltalk arrays do not hold their contents in regular Smalltalk variables. Thus, the base **DDP** algorithm provides no way to even ask for the type of an array's elements. This was satisfactory at the time **DDP** was designed, because **DDP** also provided no strategy for finding such types, either. **DDP/CT**'s source tags, on the other hand, do provide the necessary polyvariance for this analysis, and since arrays are frequently used in Smalltalk programs, **DDP/CT** also includes a new *array-element type goal*.

An array-element type goal finds the type of elements of any array in a specified array type. Ideally, the specified array type includes a source tag. In that case, the arrays whose elements are being studied are those arrays created with the specified source tag. If the array type does not have a source tag, then the solution strategy will still be followed, but most likely it will terminate quickly with a type of *Anything*.

To solve an array-element type goal, the algorithm uses a senders goal to locate all invocations of `at:put:` where the receiver might be a member of the goal's array type. Then, for each such invocation found, it posts a type goal for the second argument (i.e., the `put:` argument). Finally, it takes the union of the answers from all of those type goals and reports that union as the type of elements in the arrays in question.

5.5   Type-specific flow goals

Recall that a flow goal asks where values can flow from a specified starting location. They are used for a number of purposes, including the inverse type queries described above and finding the program locations where a particular block might be invoked. Some of the enhancements described above rely heavily on flow goals; manual inspection of early trials of **DDP/CT** suggested that the enhancements were not as effective as desired due to over-approximation in the flow goals on which the solution strategies depended.

The biggest problem appeared to be that **DDP/CT** would trace flow paths that are feasible in principle, but are infeasible for the data type of interest. For example, a variable that sometimes holds arrays that are being traced by an array-element type goal might at other times hold the value `nil`. Tracing flow through this variable would necessarily trace not only the interesting paths through which the relevant arrays flow, but also the irrelevant paths that `nil` will follow. If a message is sent to the variable, then completely different methods might be invoked when the variable holds an array versus when the variable holds `nil`; tracing flow through these later methods causes a entire subgraph of completely irrelevant program locations to be added to the potential flow from the original variable.

```
| c a other |
c := ValueHolder new.
a := c.
a contents: 'hello'.
other := ValueHolder new.
other contents: 12345.

other contents.
c contents
```

Figure 7. An example Smalltalk fragment that exhibits data polymorphism. In the first line, c, a, and other are declared as temporary variables. The ValueHolder class is instantiated twice and the two instances are assigned to c and other; a is assigned the same value as c. Thus, a and c are aliases for the same object. A string is installed into the a/c value holder on the fourth line, while an integer is installed into other's value holder on the following line. DDP/CT can distinguish these two value holders from each other and deduce that the "c contents" fetch on the final line will produce a string, as shown in Figure 8.
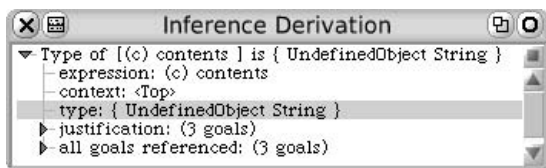

Figure 8. DDP/CT successfully infers that value holders assigned to c from Figure 7 can only hold strings and the undefined object nil. As an aside, the object can hold nil because all instance variables come into existence holding nil. DDP/CT is not flow sensitive and thus cannot determine that ValueHolder's instance variable has been initialized before contents is ever called.

The solution in **DDP/CT** is to ask a better question. Instead of simply asking about flow from a specified point, **DDP/CT** can ask about flow of objects *of a particular type* starting at a specified point. Since, in fact, every use of flow goals in **DDP/CT** is attempting to find the flow of objects in a known type, every use of flow goals can take advantage of the new facility to specify the type of objects being traced. To continue the previous example, if the analyzer is tracing the flow of arrays, then it can use a flow goal that only traces arrays. The flow-goal solver, described in subsection 4.4, is then free to ignore methods to which only nil flows.

6   Example Code Fragments

Figure 7 shows some example code that is data polymorphic. Class ValueHolder is a standard Smalltalk class used to hold an arbitrary value—it is a simple "cell" object. The internal value is set using the contents: method, and fetched using the contents method. The example code creates two value holders, storing one of them in c and the other in other. The code copies the reference in c to a, resulting in c and a being aliases to the same object. The value holder in c is given, via its alias a, the string 'hello' to hold, while the value holder in other is given the integer 12345 to hold.

This code, in isolation, uses ValueHolder in a data-polymorphic fashion; there are other methods in the standard Squeak image which use the class to contain other data types, as well. As Figure 8

```
| c a other vhclass1 vhclass2 |
vhclass1 := ValueHolder.
c := vhclass1 new.
a := c.
a contents: 'hello'.
vhclass2 := ValueHolder.
other := vhclass2 new.
other contents: 12345.

other contents.
c contents
```

Figure 9. A variation of the code in Figure 7. In this code fragment, the class ValueHolder is stored into a variable before being instantiated. DDP/CT successfully distinguishes the two kinds of value holders—those stored in c and those stored in other—just as it did in Figure 7.

```
| c a other vhclass1 |
vhclass1 := ValueHolder.
c := vhclass1 new.
a := c.
a contents: 'hello'.
other := vhclass1 new.
other contents: 12345.

other contents.
c contents
```

Figure 10. Another variation of the code in Figure 7. This time there is only one variable, vhclass1, used to hold class ValueHolder. In this case, DDP/CT fails to distinguish the two kinds of value holder created in this fragment; it infers the same types for "c contents" and "other contents". However, it does distinguish these value holders from other value holders in the program at large, ultimately inferring that both of these holders can hold only strings, integers, or the undefined object.

shows, however, **DDP/CT** successfully infers a precise type for the value held in c. It traces data flow back to the string 'hello', but ignores the infeasible data-flow path to the integer 12345.

The next two figures show variations of the code from Figure 7 in order to demonstrate an extent and a limitation of **DDP/CT**'s effectiveness. In Figure 9, the class ValueHolder is stored into variables vhclass1 and vhclass2 before being instantiated. This is an example of Smalltalk's reflective ability to manipulate classes as first-class objects themselves. This example demonstrates more clearly why "ValueHolder new" in Smalltalk is not merely a different way to write "new ValueHolder()" in Java. In this example, **DDP/CT** is still able to keep the two value holders distinct and infer that c holds only strings.

Figure 10 extends this example further and uses just one variable, vhclass, to hold the class. Both c and other are instantiated by sending new to vhclass. **DDP/CT** is unable to distinguish the two value holders in this case because it tags both of them with the singular reference to the originating occurrence of ValueHolder on line 2. Even in this case, however, **DDP/CT** is able to distinguish the two kinds of value holders in this code fragment from value holders created in other parts of the standard Squeak code base we use for our tests. Thus, **DDP/CT** infers that c holds either a string or an integer, even though there are other value holders in the program that hold other types.
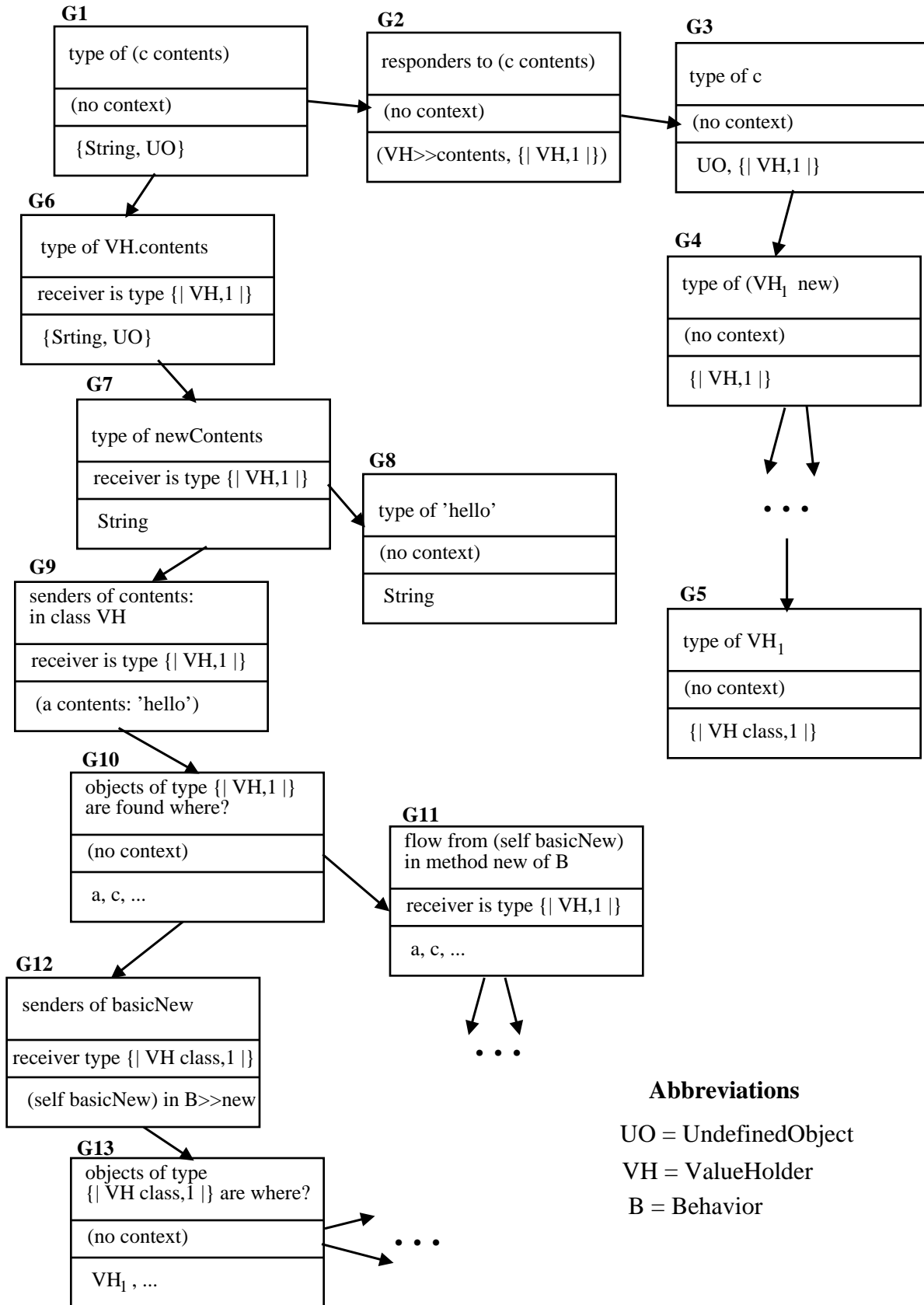
G1

type of (c contents)
(no context)
{String, UO}

G2

responders to (c contents)
(no context)
(VH>>contents, {| VH,1 |})

G3

type of c
(no context)
UO, {| VH,1 |}

G6

type of VH.contents
receiver is type {| VH,1 |}
{Srting, UO}

G4

type of $(VH_1$ new)
(no context)
{| VH,1 |}

G7

type of newContents
receiver is type {| VH,1 |}
String

G8

type of 'hello'
(no context)
String

G9

senders of contents: in class VH
receiver is type {| VH,1 |}
(a contents: 'hello')

G5

type of VH_1
(no context)
{| VH class,1 |}

G10

objects of type {| VH,1 |} are found where?
(no context)
a, c, ...

G11

flow from (self basicNew) in method new of B
receiver is type {| VH,1 |}
a, c, ...

G12

senders of basicNew
receiver type {| VH class,1 |}
(self basicNew) in B>>new

G13

objects of type {| VH class,1 |} are where?
(no context)
VH_1 , ...

Abbreviations

UO = UndefinedObject

VH = ValueHolder

B = Behavior

Figure 11. The graph of goals generated by DDP/CT when it infers a type for `c contents`, an expression in the method in Figure 12.

```
| c a other |
c := ValueHolder₁ new.
a := c.
a contents: 'hello'.
other := ValueHolder₂ new.
other contents: 12345.

other contents.
c contents
```

Figure 12. The code from Figure 7 with source tags added to the two mentions of `ValueHolder`.

```
| arr arr2 arr3 |
arr := Array new: 10.
arr at: 5 put: 'hello'.

arr2 := arr.
arr3 := arr2.

arr3 at: 5
```

Figure 13. Retrieving elements from an array. Data-polymorphic analysis is required in order for the analyzer to connect objects removed from an array using `at:` messages to objects placed into that array using `at:put:` messages.

7 Example Goal Graph

Let us now step through an example goal graph and see how source-tagged class types are threaded through **DDP/CT**. Figure 12 shows the code from Figure 7 with a couple of expressions labeled: each mention of `ValueHolder` is now tagged with its own subscript. These labels will be used in this example as source tags.

Figure 11 shows the goal graph that **DDP/CT** generates when it is asked to find a type for "`c contents`". Each box in the figure represents a goal. The top part of the box shows the query, e.g. "type of `c contents`" for G1. The middle part shows any context that should be assumed while answering the query. For G1 the assumed context is "(no context)," i.e. no assumption at all. The bottom part of each box in the figure shows the answer **DDP/CT** has found for the goal, e.g. {`UndefinedObject`, `String`} for G1.

Arrows in the figure show goal dependencies. For example, G7 depends on G9 and G8 in order to be justified. In a few places parts of the goal graph are elided from the figure. For example, G13 has a number of dependencies that are not shown, and the answer found to goal G10 has been abbreviated.

Let us now consider each goal in turn. Goal G1 is the initial goal, which asks for a type for `c contents`. This expression is a message-send expression, so **DDP/CT** relies on a subgoal that searches for the responders to the message send (G2), and subgoals for the value returned from each responding method (G6). Notice that the responder found by G3 includes a source-tagged class type, and that tagged type is passed on to the question of G6. Much of the challenge of designing **DDP/CT** consists of finding sound techniques for source tags to be passed on in this way through long chains of subgoals.

Goal G2 seeks the responders for `c contents` under no assumed context. To find the responders, a goal is posted to find the type of `c`, the receiver of the message send. The type found by that goal is $\{\!|ValueHolder, 1|\!\}$, i.e., an instance of `ValueHolder` that is associated with source tag "1." Given this type for the receiver, there is only one possible responder to the message send: the `contents` method of class `ValueHolder`, under a context where the receiver is of type $\{\!|ValueHolder, 1|\!\}$. Notice, again, that the source tag is propagated from a goal's subgoals to somewhere in the goal's answer; in this case, the tag is propagated from the type found for `c` to the responding context in G3's answer.

Goal G3 requests the type of the variable `c`. **DDP/CT** finds the type of a variable by finding the types of all expressions assigned to the variable. In this case, there is only one assignment, the expression `ValueHolder₁ new`, and so the type found for `c` is copied directly from the type found for `ValueHolder₁ new`.

The type for `ValueHolder₁ new` requires a chain of inferences that is not shown. A particularly interesting inference in the chain, however, is goal G5 which finds a type for `ValueHolder₁`. Instead of being inferred as type $\{\!|ValueHolder\ class|\!\}$ as it would be in **DDP**, it is given a type of $\{\!|ValueHolder\ class, 1|\!\}$. Thus, we finally see where source tags are initially injected into the analysis instead of propagated from one goal to another.

Goal G6 finds a type for the `contents` instance variable of class `ValueHolder`. The only assignment statement for that variable is the one in method `contents:`, which assigns the method parameter `newContents` into `contents`. Goal G7 finds a type for `newContents`, which in turn requires finding the senders of the `contents:` method. There is only one sender, and its first parameter is the literal string `'hello'`. Goal G8 finds the type of this literal, i.e. $\{\!|String|\!\}$.

Source tags pay off in goal G9. G9 finds the senders of `contents:` under the assumption that the receiver is of type $\{\!|ValueHolder, 1|\!\}$. The example code base only contains one such sender; all other statements that invoke this `contents:` method invoke it on receivers with different source tags. Without source tags, the answer to G9 would include a much larger number of senders.

Goal G10 finds all expressions that hold an object of type $\{\!|ValueHolder, 1|\!\}$. It is a subgoal of G9 and is used to find those senders of `contents:` where the receiver has source tag 1. **DDP/CT** creates goal G12 to find senders of `basicNew` where the receiver is of type $\{\!|ValueHolder\ class, 1|\!\}$. That is, to perform an inverse type query on a regular class, **DDP/CT** performs a senders query on `basicNew` for the associated metaclass.

Goal G12 is a senders query where the assumed receiver type $\{\!|ValueHolder\ class, 1|\!\}$. To answer G12, **DDP/CT** creates goal G13 to find all expressions that hold an object of type $\{\!|ValueHolder\ class, 1|\!\}$.

Goal G13 is an inverse type query that finds all program locations holding a value of type $\{\!|ValueHolder\ class, 1|\!\}$, a type for a metaclass. **DDP/CT** immediately includes expression `ValueHolder₁` because it evaluates to `ValueHolder class` with a source tag of 1. **DDP/CT** immediately *excludes* all other `ValueHolder` expressions, e.g. `ValueHolder₂`, because they have a different source tag. **DDP/CT** additionally considers alternatives ways class object can enter the computation in Smalltalk, but in this case there are none and the subgoals are elided from the figure.

Figure 14. The analyzer succeeds on the example in Figure 13.

```
| p1 p2 |
p1 := Array new: 3.
p1 at: 1 put: 2.
p1 at: 2 put: 3.
p1 at: 3 put: 0.

p2 := Array new: 3.
p2 at: 1 put: 3.
p2 at: 2 put: 4.
p2 at: 3 put: 1.

^p1 dotProduct: p2
```

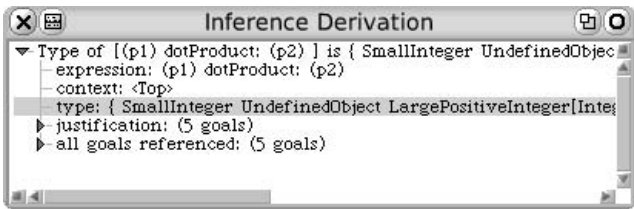**Figure 15. Data-polymorphism occurs in numeric array computations.**

Figure 16. The analyzer succeeds on the example in Figure 15.

8 Collection types

Data-polymorphic analysis is especially useful when it is applied to resolving separate uses of collection types. A simple example is shown in Figure 13. The code creates an array, adds the string 'hello' to it, and then retrieves that same string. The analyzer succeeds in this case, as shown in Figure 14. The analyzer uses source tags to connect the at: message-send on the last line of the example to the at:put: message-send on the third line of the example, while ignoring the other 1706 senders of at:put: in the same code base.

A more useful and sophisticated example is shown in Figure 15. In this example, we create two numeric vectors, then compute their dot product. The dotProduct: method, not shown, includes a number of senders to at:. **DDP/CT** can connect those senders to the senders of at:put: in Figure 15 using class tags, and determine that all of the arithmetic operations the dotProduct: method uses will be applied to integers. The result produced by **DDP/CT** is shown in Figure 16.

9 Multi-level source tags

Factory design patterns [6] present an extra challenge to data-polymorphic analysis. A typical factory method is shown in Figure 17. This method provides a useful level of indirection—subclasses might override this method, and different platforms might replace the method outright. Unfortunately, the very indi-

```
Platform>>makeHolder
    ^ValueHolder new
```

Figure 17. A typical factory method, makeHolder, for class Platform. This kind of indirection is useful in many circumstances, including the possibility that different platforms will implement the method to use a different value-holder class. Unfortunately, all callers of this method will receive a ValueHolder with the same source tag: the single mention of ValueHolder in the makeHolder method.

```
| vh1 vh2 |
vh1 := Platform makeHolder.
vh1 contents: 'hello'.

vh2 := Platform makeHolder.
vh2 contents: 123.

vh1 contents.
```

Figure 18. An example usage of the factory method from Figure 17. In this example, the inferencer as described so far fails to distinguish separate container objects, because both holders are given the same source tag.
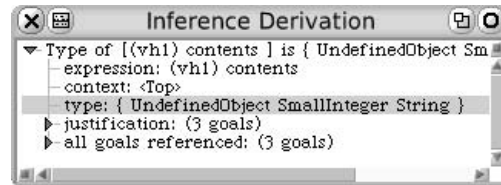

Figure 19. The analyzer merges flow through the two different holders in Figure 18, and so reports that vh1 can hold both integers and strings.

rection that motivates the design pattern overloads the necessarily-finite abstraction of class tags: all value holders created by the makeHolder method are given the same source tag. Thus, the central approach of this paper, as described so far, is insufficient to distinguish separate uses of objects created by factories.

A sample use of this factory method is shown in Figure 18. Since the same source tag is used for the value holders held by both vh1 and vh2, data flow through the distinct holders is intermingled as shown in Figure 19.

This example points to a solution, however. Notice that, while the vh1 and vh2 value holders are both associated with the single mention of the ValueHolder class in the Platform factory method, they access that method through separate mentions of class Platform. If there were a way to tag the ValueHolder references with the mention of Platform instead of the mention of ValueHolder, then the two variables' value holders could be discriminated by the analysis.

This can be accomplished by generalizing source tags into flow positions. A flow position can include both a pointer to an expression in the program plus a context under which the expression was evaluated. The context can include a type for the surrounding method's parameters and for the current receiver object. The type of the receiver object, in turn, can be another source-tagged class type, completing a recursion. Thus, generalizing source tags into flow positions allows the system to apply multiple tags to the same object.
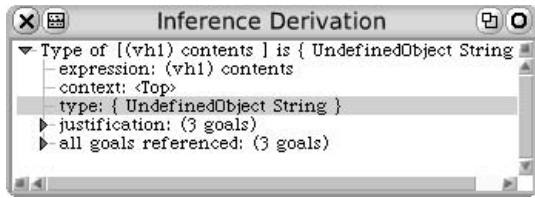
46

Figure 20. Using multi-level source tags on the example from Figure 18, it is possible to distinguish objects that are created via a factory object.

A maximum number of tags—i.e., traversals through the recursive cycle of tags to contexts to types to tags—must be chosen to keep the data-flow lattices finite. Choosing a maximum tagging level of 1 yields an analyzer equivalent to one using simple source-tagged class types. A level of 0 gives a system that does not use source tags at all. A level of 2 is sufficient for the example of Figure 18, resulting in a precise type inference, as shown in Figure 20.

10   Related work

A large number of alias-analysis algorithms partition allocated objects using "allocation sites" [9]. An allocation site is typically an invocation of `new` or `malloc()`. For example, the **DCPA** algorithm by Wang and Smith partitions objects by which `new` statement allocates them [23]. **DCPA** has also been adapted to the Resilient Smalltalk environment[5] by von der Ahé [22]. The inferencer of von der Ahé is also able to separate analysis of an object's lifetime to times before and after the object's initialization code finishes; in many cases, this separate analysis avoids the imprecision caused by variables being auto-initialized to `nil`.

As discussed earlier, partitioning by `new` statements alone is unhelpful by itself for Smalltalk code. **DCPA** additionally discriminates allocation sites using context, or contours, and its context-discrimination mechanism depends on distinguishing between "CPA safe" and "CPA unsafe" methods. This additional discrimination is used by von der Ahé to obtain effective data-polyvariant analysis in Smalltalk. The distinction is computed by marking some methods as unsafe, in particular those that create new objects, and tracing backwards through the call graph: any method that invokes an unsafe method is also unsafe. The remaining methods are marked as safe. While this computation is effective in an abstract-interpretation framework, where the full program is analyzed, it appears impractical in **DDP**'s demand-driven framework. Only a small fraction of the program is analyzed in each execution of a **DDP**-based algorithm, and determining the safe vs. unsafe designation would require analyzing a substantial extra portion of the program.

A type-inference algorithm crafted by Oxhøj, Palsberg, and Schwartzbach also partitions objects by allocation site [13]. **DDP/CT** contributes a novel analog to allocation sites, source tagging of class types, which is effective in highly dynamic languages with only one allocation site.

Plevyak and Chien describe an adaptive algorithm that often avoids using type tags when they would not be able to refine the analysis [14]. This approach speeds up the algorithm with no loss in precision. **DDP/CT** is less sophisticated and uses source-tags generously even when they are not needed. This profligacy of potentially su-

perfluous precision is mitigated, however, by **DDP/CT**'s ability to focus effort on a relatively small portion of the program. **DDP/CT** may not happen to analyze a large number of uses of the same class at all in the sparse elements of the program it traverses for a given request, independent of whether or not their analysis could have been merged without loss of precision.

11   Future work

As we've seen in our examples, the single-level source-tagging abstraction does not distinguish distinct instances of higher-level collections, such as the `OrderedCollection` and `Set` classes, that are built on top of underlying arrays. The multi-level source tags described in section 9 provide enough information to the analyzer to distinguish these uses. We are in the midst of a detailed study of **DDP/CT** effectiveness on code samples drawn from real-world code bases such as the Squeak run-time system. Our tests are intended to evaluate the practical efficacy of source-tagged class types by comparing the performance of the algorithm—both for speed and precision—using a variety of levels of source tagging, including zero levels (i.e., no source tags).

Measuring speed and precision across variations in the abstraction is particularly interesting in the case of **DDP/CT**. It is typically the case in program analysis that increasing the precision of an abstraction causes the algorithm to run slower. However, we have found in our initial trials that the increased precision of source tags has a strong focussing effect on the (dynamically determined) control-flow links along which **DDP**'s goal-directed, demand-driven search proceeds. That is, source tags allows the analysis to eliminate large portions of the code base from consideration that would otherwise be conservatively dragged into the analysis. It appears that instead of a precision/speed tradeoff, increasing the precision *also* improves the scalability and speed of the analysis. The tests we are currently conducting will allow us to evaluate this possibility in a quantitative manner.

12   Conclusion

Data polymorphism is a long-standing issue with data-flow analysis in higher-order, dynamic programming languages. The problem is made even more difficult in languages that reflectively allow classes to be reified as first-class objects. **DDP/CT** is a solution to this classic problem, extending the basic **DDP** analysis with a simple form of polyvariance that is able to resolve source-distinct instances of data-polymorphic classes. Despite this extra precision, it retains the scalability of the original **DDP** analysis. Our implementation demonstrates that the analysis can handle the full complexities of a real-world language, even one with the extremely dynamic, reflective features of Smalltalk.

13   References

[1] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, Jr. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised[5] report on the algorithmic language Scheme. *SIGPLAN Notices*, 1998.

[2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

---

[5]`http://www.oovm.com`

[3] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1995.

[4] 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.

[5] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Boston, MA, 1996.

[8] David Grove, Greg Defouw, Jeffrey Dean, and Craig Chambers. Call-graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1997.

[9] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Snowbird, Utah, 2001.

[10] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1997.

[11] Alan C. Kay. The early history of Smalltalk. In *The second ACM SIGPLAN Conference on the History of Programming Languages*, pages 69–95. ACM Press, 1993.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.

[13] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 329–349, 1992.

[14] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, pages 324–340, 1994.

[15] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data-flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

[16] Olin Shivers. Control-flow analysis in Scheme. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[17] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[18] Olin Shivers. The semantics of Scheme control-flow analysis. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 190–198, 1991.

[19] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scala-

bility. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[20] S. Alexander Spoon and Olin Shivers. Semantic navigation of large code bases in higher-order, dynamically typed languages. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, Pittsburgh, Pennsylvania, November 2005.

[21] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, pages 292–305, New York, NY, USA, 1999. ACM Press.

[22] Peter von der Ahé. Applications of concrete-type inference. Master's thesis, University of Aarhus, 2004.

[23] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. *Lecture Notes in Computer Science*, 2072:99–117, 2001.