Demand-Driven Type Inference with Subgoal Pruning

S. Alexander Spoon
Olin Shivers

Sun, 05 Mar 2006
(revision 1639)

Summary

Highly dynamic languages like Smalltalk do not have much static type information immediately available before the program runs. Static types can still be inferred by analysis tools, but historically, such analysis is only effective if context sensitivity is abandoned or if focus is restricted to smaller programs with only tens of thousands of lines of code.

This paper presents a new type inference algorithm, **DDP**, that is effective on larger programs with hundreds of thousands of lines of code, and that integrates elegantly with integrated development environments. The approach of the algorithm borrows from the field of knowledge-based systems: it is a demand-driven algorithm that sometimes prunes subgoals. The algorithm is formally described, proven correct, and implemented. Experimental results show that the inferred types are usefully precise. A complete program understanding application, Chuck, has been developed that uses **DDP** type inferences.

This work contributes the **DDP** algorithm itself, a semantics of Smalltalk, a new general approach for analysis algorithms, and experimental analysis of **DDP** including determination of useful parameter settings. It also contributes an implementation of **DDP**, a general analysis framework for Smalltalk, and a complete end-user application that uses **DDP**.

Contents

Chapter 1

Introduction

1.1 Overview

Dynamic programming languages give a tight interface between programs and humans. They do so in part by removing the need to restart a program whenever the human requests changes to be made. The result is an interface like Smalltalk [8, 42] or the Lisp Machine [45], interfaces where the human is more like a sculptor molding clay than an operator submitting punched cards. Such interfaces share a similarity with mature operating systems, where users may make many changes without rebooting the entire computer. Users of a dynamic language, similarly, can make many changes without rebooting the entire running program.

These dynamic interfaces must tolerate programs that are less than pristine. In particular, the languages must have very flexible type systems in order to avoid chicken-and-egg problems whenever a programmer tries both to change the type of some variable and to update the locations the variable is used. This type-checking challenge is so great that most dynamic languages include no type checker at all. As a result, programmers in dynamic languages can make changes more readily, but they have less automatic information about the programs they have created.

Type checkers, however, give useful type information. Such types can be used for program understanding [24], for dead code removal [2], and for improved compilation [32, 60]. By giving up a type checker, dynamic programming environments seem to sacrifice these good static tools.

There is another source of type information, however: program analysis. Specifically, type inference. Type-inference algorithms can analyze a program and produce correct statements about the types that portions of a program will have when the program executes, even in environments that do not insist on all programs type checking.

The type-inference problem is challenging. Such algorithms must successfully process arbitrary programs, in the full generality that programmers are allowed to use in a dynamic language, in contrast to a type checker that is allowed to reject sufficiently difficult programs. Such an algorithm must, for many languages, contend with data flow and control flow depending on each other. Such algorithms can infer better types when they repeatedly analyze the same expressions under multiple assumed execution contexts, yet history shows that they must be careful not to analyze under too many contexts or they will require too much memory (and thus time) to be practical.

This work describes a new type inference algorithm and shows that it is effective. Specifically:

> Demand-driven algorithms that prune subgoals can infer types that are correct, that are usefully precise, and that differ depending on calling context, in Smalltalk programs with hundreds of thousands of lines of code.

1.2 Problem details

The problem addressed in the present work is to infer types in large Smalltalk programs without giving up on context sensitivity. This section describes several aspects of the problem.

1.2.1 Large programs

Type inference is an old problem, and there are now effective algorithms for programs of up to tens of thousands of lines of code, even with all of the other problem constraints described below. Therefore, the present work focuses on larger programs of at least one hundred thousand lines of code. When we write of "large programs," we mean programs with at least one hundred thousand lines.

1.2.2 Sound upper bounds

The correctness requirement of the present work, defined in detail in Chapter 6, is that inferred types must be sound upper bounds. Consider a type judgement such as, "`foo` holds an `Integer` or a `Float`." The correctness requirement is that every value held by the variable `foo` as the program runs is either an `Integer` or a `Float`. It is acceptable to have extra options, for example if `foo` actually only holds `Integer`'s and never `Float`'s. It is not acceptable for `foo` to hold `Fraction`'s.

Potential uses do not need to be reported. For example, the above judgement is correct even if the code will function correctly when `foo` is bound to a `Fraction`. As a result, a library is allowed to have different types inferred when it is used by different programs. In short, the present work finds actual uses instead of potential uses.

1.2.3 All programs accepted

The goal of the present work is to accept all programs. It is a pure *program analysis*, producing information about an existing program, as opposed to a *program verification*, which attempts to verify that the program matches some specification—in this case, the specification that no type error occurs when the program runs [57]. Program verification cannot succeed on an arbitrary program. For typical problems, verification cannot even succeed on all programs that match the specification—otherwise, the algorithm would provide a solution to the Halting Problem. Verifiers, therefore, must always reject some programs and must typically reject even some satisfactory programs. The assumption in the present work is that too much code already exists to allow this kind of rejection. The present work applies to arbitrarily objectionable programs.

The correctness requirement described above, "sound upper bounds," follows from this choice. Many other researchers study a stronger correctness requirement, that no type errors occur at run time, but such researchers must allow some programs to be rejected. This stronger property has two parts, *progress* and *preservation* [52], of which the present work only guarantees preservation. A type system guarantees progress if, whenever the types are correct, the program will continue executing. A type system guarantees preservation, if whenever the program continues executing, the types remain correct. In the present work, type information is correct so long as the program continues executing, but the program might nonetheless stop executing at any time.

1.2.4 Concrete types

Types, in the present work, are an abstraction over the concrete behavior of a program, and abstraction has an inherent tradeoff between brevity and detail. Extremely abstract types concisely describe program behavior program, but they lose detail. Extremely concrete types provide great detail about the program, but they lose brevity.

The present work studies relatively concrete types, such as “an Integer or a Float”, instead of relatively abstract types, such as “a function from (α,β) tuples to α’s”. The precise type system is described in Chapter 6. In general, the strategy is that followed by Agesen [2]. Concrete types are useful for finding control flow information, which in turn is useful for many other program analyses. Overall, concrete type inference is a stepping stone to other analyses.

1.2.5 Higher-order languages

Higher-order languages are desirable, but they make analysis more difficult. In particular, higher-order languages have subroutine calls that semantically are bound at run time. Object-oriented languages dynamically bind message sends to methods, while functional languages dynamically bind function calls to functions. Classic data-flow algorithms for first-order languages [5] cannot be used as they are on higher-order languages, because such algorithms presume that a control-flow graph is easily computable before starting the analysis proper.

A conservative control-flow graph may still be computed through program analysis. This computation, however, requires type information in order to be precise. The two problems are thus intertwined: finding type information requires finding control-flow information, and vice versa.

1.2.6 Smalltalk

It is expected that the present work is applicable to a variety of programming languages. In order to make progress, however, a specific dynamic programming environment has been studied initially. Smalltalk has been chosen due to several advantages: it is used for larger programs; it is a small language and thus convenient work with; and it includes the higher-order constructs of message sending and higher order functions. Additionally, typical Smalltalk code makes exceptionally heavy use of run-time binding. Even the conditional and looping constructs are implemented with higher order functions instead of being in the syntax. Smalltalk programs thus stress a program analysis to an exceptional degree. An algorithm effective in such an extremely dynamic language is likely also to be effective in other, less dynamic languages.

Study of type inference in other dynamic languages is left for future work as described in Chapter 12.

1.2.7 Context-sensitive analysis

The present work limits attention to context-sensitive type inferencers with directional data flow (and thus that are not based on unification—these terms are described in Chapter 2). Such algorithms are widely agreed to produce more precise information about a program compared to other type inferencers, but they are also widely rejected for use in large programs due to expected scalability difficulties. It is not necessary to not reject such algorithms, however, and indeed the present work demonstrates a context-sensitive inferencer that scales.

Our project *deeply* studies one context-sensitive inferencer instead of *broadly* studying a variety of inferencers including context-insensitive ones. Adjusting the existing alternative inference algorithms for Smalltalk requires substantial effort—it is more difficult than simply adjusting for a different syntax. As one example, the expressions “Morph new”, “HtmlDocument new”, and “OrderedCollection new” would, without care, all be merged by the analyzer and given the same (large) type. Smalltalk is simply a very dynamic language; `new` is a method in the library instead of syntax. Given the success of unification-based algorithms in Cecil [20], it is likely that such algorithms can be adjusted to work in Smalltalk. Since it is not expected that they generate information as precise as context-sensitive algorithms generate, this approach is not pursued in the present project and thus it is left as an open research area.

The choice of studying context-sensitive analysis with directional data flow has two major benefits. First, such analyzers have performance characteristics appropriate to the application area of interactive programming tools. While it is likely that unification-based algorithms can be effective in Smalltalk, it is less clear that they can produce results at the interactive speeds described in Chapter 10 and particularly section 10.7.

The second benefit is that the work achieves a wider impact. The analysis approach described in this document should also be effective in less dynamic languages such as Java, and thus the present work revitalizes context-sensitive analysis in general.

1.3 How to read this document

The overall structure of this document is:

1. Informal development.
2. Formal work.
3. Independent chapters exploring one topic each.

The informal development begins with the careful problem statement of this chapter, continues with a review of the history of type inference in dynamic languages, uses that review to motivate the structure of **DDP**, and then describes **DDP** in detail. This work constitutes Chapter 1 through Chapter 4.

The formal work of Chapter 5 through Chapter 8 first formalizes three things: the language semantics, the domains of data-flow facts, and the justification rules used to justify inferred facts about a program. The formal work culminates in a proof that these definitions correspend in the appropriate way: that justified judgements are correct.

The chapters after the proof of correctness each stand alone. There is a description of Chuck, a program-understanding tool based on **DDP**. There is a description of an experiment that measures **DDP** in practice. There are a few recommendations for dynamic languages of the future to better support type inference. Finally, there is discussion of future work in this line of research, including a description of a beginning on this work, and some concluding remarks.

Different readers will want to focus on different parts of this document. Some suggestions are given below. If you encounter unfamiliar terms or function names due to skipping chapters, try the index; all functions and defined terms have an index entry.

If you want to implement a type-inference tool, then you are probably most interested in the workings of **DDP** and its performance envelope. You should focus most closely on the informal development of the first four chapters of the work, and on Chapter 10 to gain some intuition about how to tune the main parameter of **DDP**. Chapter 13 has information about a promising direction of development for type inference. If you are not familiar with Smalltalk, you should also skim Chapter 5 in order to learn the language syntax that is being used throughout this document.

If you are a program analysis researcher, then you are probably most interested in the differences between **DDP** and other program analyses. You should focus on Chapter 2, Chapter 3, and Chapter 13. Additionally, you may be interested in Chapter 6, which builds on existing work to define refined domains for describing context-sensitive data-flow facts.

If you are a language designer, then you are most interested in how type inference in dynamic languages is progressing and on how language design makes analysis more or less effective. You should focus on Chapter 1 and Chapter 10, as well as skimming Chapter 2, to gain a view of the status of type inference as of the time of this work. Additionally, you should read Chapter 11 to see recommendations stemming from this work for the development of future dynamic languages.

Chapter 2

Related work

Type inference in dynamic higher-order languages has been studied for decades. This chapter describes this related work from several different perspectives.

2.1 Related problems

Several analysis problems are closely related to type inference. Type-inference enthusiasts should be aware, while reading the literature, that algorithms for a related problem often include many ideas relevant to type inference. In fact, many algorithms which directly solve the related problem also solve a bona-fide type-inference problem along the way. This section describes several related problems that have been studied and should be considered even by those ultimately interested in type inference.

The problem examined in the present work is *type inference* [2, 12, 27, 72, 20, 66]. This problem also goes by the names *type determination* [66], *concrete type inference* [2], and *class analysis* [20]. The problem, from this perspective, is to analyze a program and predict what type of values the variables or expressions of the program will hold when it runs. The *inference*, *determination*, or *analysis* part means that the program is assumed to have no type annotations on the variables, implying that the analyzer needs to infer types where none are explicit. The *concrete* or *class* part means that the kind of types being inferred are sets of runtime values. That is, they are types such as “Integer, Float, or Fraction,” as opposed to abstract types such as, “an expression that has no side effects”. There is no exact boundary between abstract types and concrete types, but most would consider both sets of classes and the types inferred in the present research as relatively concrete.

A related problem is *data-flow analysis* in general [11, 22, 57]. To infer types, the algorithms typically find paths through which values flow from one part of the program to another. For example, if they see a statement `x := y` in the program, they note that there is a path from `y` to `x`. Any types that arrive in `y` can flow on to `x` as the program executes. Conservatively approximating the resulting flow is the problem of data-flow analysis. Inferring types usually involves an algorithm that is sufficient to perform data-flow analysis in general, and vice versa.

Finally, *control-flow analysis* [60] is a related problem. *Call-graph construction* and *call-graph extraction* are examples of control-flow analysis in object-oriented languages. In general, control-flow analysis predicts the order in which parts of a program will execute. In higher-order languages, where there are late-binding constructs such as message sending and first class functions, finding precise control flow requires predicting types as well. To find the control flow for a message send, one must predict the classes to which the receiver might belong; to find the control flow for a function invocation, one must predict which functions might flow to the function expression at the call site. In both cases, finding precise control flow requires also finding concrete types along the way. Similarly, finding precise types in a higher-order language requires predicting how the late-binding operations will be bound, thus showing that type inference is the same problem as precise control flow. Do note that less precise control-flow algorithms do not need to find types: they make conservative estimates of the late-binding operations and thus do not need to find type information. The fastest algorithms described in the survey by DeFouw, et al., make just such a trade off [20].

2.2 Applications

Type inference is usually studied in order to enable some specific application. All existing type-inference techniques are useful for all of these applications, though different applications will prefer the use of different techniques. Some applications prefer a fast type-inference algorithm that finds types quickly enough that they can be used in interactive tools, while other applications only require that the inferencer be fast enough to find the types overnight. Some applications need precise types to be useful at all, while others can fruitfully use types that are not very precise. Some applications prefer a type inferencer that can be focused to find types for one specific portion of a program, while for others the inferencer may as well analyze the entire program.

My motivating application is *program understanding* [24]. Inferred types can help a programmer who is trying to understand the internal workings of a particular program. The inferred types are directly useful themselves, and they also help program-understanding tools such as diagram generators and static debuggers. Program understanding applications prefer those type-inference algorithms that run relatively quickly, as well as those that can be focussed on the portion of a large program that the programmer is currently studying.

Another common application is *program transformation*, including transformations to make a program run more quickly (*compiler optimization*) [10], and transformations to remove portions of a program that are not needed (*dead-code removal*) [3, 68]. Transformation for speed typically prefers a type inferencer which can be targeted to a module or less at a time in order to support separate compilation. Removing unused code requires an inferencer that can efficiently analyze an entire program. Neither kind of transformer has special speed requirements; it is a useful tool even if it must run overnight instead of running interactively.

Third, there are interactive programming tools that are more effective if they have better type information. A refactoring browser [56] can make more fine-grained refactorings if it has better type information. For example, if a user requests that a particular method be renamed, a refactoring browser must additionally rename some other same-named methods in parallel; type information can reduce the number of such additional methods that need to be renamed. The basic name-completion commands of an interactive text editor need a shorter prefix from the user if type information is available to lower the number of names that are relevant in a particular context. Such tools prefer a type-inference algorithm that runs at interactive speeds and that can be targeted at specific parts of the program.

A final application is *error detection* [64]. Type inference can be used to find potential locations in the program where, for example, a message-send expression might fail to bind to a method (i.e., a Smalltalk "does not understand" error). Error detection requires a highly precise type inferencer, but it does not require that the inferencer be targeted at a portion of a program nor that it run especially quickly.

2.3 Aspects of existing algorithms

This section discusses several aspects of existing type-inference algorithms. For each aspect, the section describes the history of proposals for that aspect and then gives, from the point of view of the present work, the state of the art on that aspect.

This approach seems more helpful to the reader than a description of individual projects in detail. Future algorithms will be built by considering those aspects, not by mimicking individual projects, and thus an understanding of the individual aspects is important. Nevertheless, extensive reference is made to individual projects. Readers can, whenever they are interested, assemble these references into a complete picture of each project from the point of view of the present project.

2.3.1 Algorithm frameworks

There are three common algorithm frameworks used for type inference: abstract interpretation, constraints, and demand-driven analysis. This section describes gives an overview of those three approaches.

Abstract interpretation

The *abstract interpretation* framework treats analysis as an abstraction of execution [19, 40]. That is, whereas the normal interpreter for a programming language computes with real program values and real variable bindings, an abstract interpreter computes with abstract values—such as *types*—and abstract variable bindings.

Formally, a regular interpreter might be described with equations like $E(e) = v$, meaning that evaluating (E) the expression e yields the value v. An abstract interpreter is described with equations more like $\hat{E}(e) = t$, meaning that abstract interpretation ($\hat{E}$) of the expression e yields something of type t. Such an analysis is correct if, for every e, $E(e)$ is indeed a value of type $\hat{E}(e)$. In a word, the abstraction should be *consistent* with the concrete semantics.

In order to support more analysis problems, often a *non-standard semantics* is used instead of the usual language semantics. For example, if one wishes to find feasible call-graph edges, then one might begin by defining a non-standard semantics E' such that $E'(e) = (v, c)$ determines not only the value v that is computed by e, but also the list of call graph edges c that are invoked in the course of computing that value. An analyzer is then defined using a *non-standard abstract semantics (NSAS)*, and the analyzer is correct if the NSAS corresponds to the non-standard semantics. Since the correctness of such an abstract interpretation depends on the choice of non-standard semantics, the non-standard semantics in effect *defines* the analysis problem.

Shivers used the abstract-interpretation framework to describe an entire family of type-inference algorithms for Scheme [60]. The algorithms within the family are differentiated by the following two parameters:

- *Abstract values*, or *types*, are an abstraction of program values.
- *Abstract contours*, or *context*, are an abstraction of control and environment context. Context is discussed further in subsection 2.3.2.

Jagannathan and Weeks later describe a similar framework that includes other algorithm parameters [39]. Sharir and Pnueli also use abstract interpretation in their early description of interprocedural data flow [57]. Garau uses abstract interpretation to implement his Smalltalk type inferencer [27].

Constraints

The *constraints* framework describes algorithms as *generating* a number of constraints from the program and then *solving* those constraints to find information about the program. Constraints are usually generated by simple syntax analysis. For example, every statement of the form $[\![x := y]\!]$ might generate a constraint of the form *t_x is a supertype of t_y*, where t_x and t_y are variables representing a type. A solution to the constraints is an assignment for all of the analysis variables (t_x, t_y, ...) such that all of the constraints are satisfied.

Constraints come in a variety of forms, and each form leads to a different method of solution. Constraints such as $t_x \sqsubseteq t_y$, "t_x is a subtype of t_y," lead to iterative solutions similar to those used in classic intraprocedural data flow. Conditional constraints, such as $t_r : T \Rightarrow t_x \sqsubseteq t_y$, capture data flow in higher-order languages. In this example, the constraint claims that if t_r includes type T, then the constraint $t_x \sqsubseteq t_y$ becomes effective. Such constraints capture new data-flow paths becoming feasible as control-flow paths become feasible. Equality constraints, such as $t_x \equiv t_y$, lead to the unification-based algorithms discussed further in subsection 2.3.4.

Implementations take considerable liberty within the general constraints framework. Frequently, constraints are not represented explicitly; since constraints are typically closely based on program syntax, the constraints in many algorithms may as well be inferred as the analyzer progresses instead of in a separate constraint-generation phase. Additionally, even when constraints are explicit in the implementation, they are not always generated until there is reason to believe they will influence the final result. In particular, a highly context-sensitive algorithm frequently has many conditional constraints that never become effective.

Constraints can be simplified considerably without affecting the solution to those constraints. Some researchers have obtained substantial speed improvements by performing such simplifications before proceeding to solve the constraints [54, 25, 6].

A large number of data-flow research projects use the constraints framework, including the work of: Kaplan and Ullman [41]; Suzuki [64]; Henglein [36]; Oxhøj, Palsberg, and Schwartzbach [51]; Emami [23]; Agesen [2]; Steensgaard [63]; DeFouw, Grove, and Chambers [20]; Flanagan and Felleisen [25]; Tip and Palsberg [69]; Aiken [6]; Wang and Smith [72]; and von der Ahé [71].

Demand-driven analysis

Demand-driven algorithms are organized around *goals*. A client posts *goals* that the algorithm is to solve, and the algorithm itself may recursively post more goals—*subgoals*—in order to solve the initial goals. The goal-subgoal relationship may be cyclical: a goal can be a subgoal of one of its subgoals. When there is a cyclical subgoal graph, the algorithm typically update goals repeatedly until every goal is consistent with its subgoals.

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 [55, 22, 4, 35, 21].

There are two primary advantages of a demand-driven analysis over an *exhaustive analysis*. First, a demand-driven algorithm analyzes a subset of the program for each goal. If only a small number of goals are needed, and only a limited portion of the program is analyzed while solving each goal, then a demand-driven algorithm can finish more quickly than an exhaustive algorithm. The exhaustive algorithm must analyze the entire program (or at least the live portion of it), while a demand-driven algorithm can focus on the parts of the program relevant to the initial goals. This advantage is particularly important for interactive program-understanding tools, where users ask the tool for information on whatever code they are currently viewing.

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 subgoals and accept lower precision in the target goal. This idea is explored in the next chapter.

The primary disadvantage of a demand-driven analysis is that it only finds information about those constructs for which goals have been posted. If a client is in fact interested in information about all constructs in an entire program, then it must either post an enormous number of goals, or it must run the analysis many times with different initial goals. Thus a demand-driven analysis is typically slower than an exhaustive analysis if the client does, in fact, want information about the entire program.

2.3.2 Context and kinds of judgements

Type-inference algorithms typically produce one type *judgement* for each variable of a program.[1] Algorithms differ widely, however, in the judgements they process before producing their final results. When an algorithm processes multiple judgements for each variable, the algorithm is called *context-sensitive* or *polyvariant*. Other algorithms, at the opposite end of the spectrum, process judgements that each describe multiple variables. In the middle of the spectrum are algorithms that process exactly one type judgement per variable. Examples are Kaplan and Ullman's algorithm [41] and 0-CFA[60].

At one end of the spectrum, context-sensitive algorithms process multiple judgements for each variable of the program. The judgements for a particular variable are distinguished by their *contexts*. A context, broadly, is some assumption about the state of execution. A judgement only applies when its context matches the state

[1]For clarity of exposition, algorithms are described in terms of assigning types to variables, even though many algorithms assign types to other syntactic elements such as expressions, functions, classes, or methods. The distinction is irrelevant for the present chapter.

of execution. When the context does not match, the judgement states nothing and is trivially correct, much as an implication in logic is vacuously true whenever its assumption is false.

A judgement with a specific context applies only to a small portion of possible execution states. To produce final judgements with no context, the algorithm must analyze each variable under enough contexts that all possible execution states are matched by at least one of the contexts. If the algorithm uses restrictive contexts that only match a small portion of execution states, then the algorithm must analyze each variable under a large number of contexts; likewise, if the algorithm uses broadly applicable contexts, then it needs to analyze under fewer contexts per variable. Specific contexts tend to find more specific final information, but also tend to require more total execution time due to the increased number of judgements that are studied [32].

One widely studied kind of context is the *call chain* [57]. A call chain specifies which call statements are at the top of the call stack. 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 number of call statements in a chain is typically limited by a constant that is a parameter of the algorithm. For example, an algorithm might use call chains of length 4. The number of contexts per variable is at worst 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** [60] and Emami's points-to analysis [23].

Another widely used kind of context is the *parameter-types context*. A parameter-types context specifies the types of parameters of the currently executing method. For example, "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`."

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 [32].

The terms *context* and *calling context* are common [57], but other terms have been used as well. Agesen discusses multiple *templates* of a method, where the templates differ in what this document calls context [2]. Shivers' mathematical formulation of control-flow analysis in Scheme defines context using *abstract contours* and *contour-selection functions* [60].

While the present project uses **CPA**-style parameter-types contexts, this aspect of type inference is not settled. One call-graph survey[32] gives empirical results about their effectiveness in Cecil and Java, with algorithms using a traditional *control process* (see Chapter 3). However, more empirical research is needed before it is possible to characterize the different kinds of context under broader circumstances, especially in light of the new control process described in the present work.

Finally, at the opposite end of the spectrum from context-sensitive algorithms, there are algorithms that process judgements that each apply to multiple variables. For example, the **XTA** algorithm makes judgements of the form, "any variable in method *m* is of type *t*" [69]. Tip provides evidence that **XTA** is effective for Java programs, but this author knows of no attempt to use this approach in a language without static types. Perhaps, static types counteract the loss of precision due to mixing multiple variables in the same judgement. Without static types, the approach may be too imprecise to yield useful results. To date, no empirical evidence is available to decide.

2.3.3 Program expansion before analysis

Program *expansion* is an approach, not used in the present work, for gaining context-sensitive analysis without using context. The approach is to duplicate portions of the program before the main analysis executes. The duplication increases the size of the program that the main portion of the analyzer processes. When expansion is used, the analysis as a whole can find context-sensitive information even if the main analysis is not context sensitive.

Expanding calls is one way to expand programs before analysis [51]. For each method name *m* and each call statement s that invokes a method named m, a new method name m_s is computed. All methods named m are given an exact duplicate for each such m_s except that the name has been changed from m to m_s. All message-send statements s that invoke a method named m are rewritten to invoke m_s instead of m. This transformation yields a program that behaves equivalently to the original program. However, each duplicate of a method m may now be analyzed independently. The analysis becomes context-sensitive. The results are equivalent to using call-chain contexts with chains of length 1.

Expanding away inheritance is another way to expand object-oriented programs before analysis [31, 51]. Each method is copied to each class that inherits the method. As a result, each method is analyzed multiple times, once for each possible class of the receiver. The results are equivalent to using parameter-types context, where the receiver type of a context is a single class and all parameter types of a context are the all-inclusive type.

Context, in general, is more flexible than expansion and is more convenient to discuss. Notably, at least some work treats expansion as a formalism and uses an implementation that only duplicates methods on demand [31]. The present work uses context instead of program expansion.

2.3.4 Unification-based data flow

Some algorithms consider the direction of data flow while others do not. The latter algorithms are said to use *unification*, because they proceed by equating (unifying) types with each other. Most of the algorithms cited in this chapter use directional data flow because it is more precise, but unification-based analysis can be executed more quickly.

Notable unification-based data-flow algorithms include those of Henglein [36], Steensgaard [63], and DeFouw et al. [20].

2.3.5 Stopping early

The theoretical framework varies among type inference algorithms. Early algorithms such as Kaplan and Ullman's begin with trivially safe judgements such as "variable x has type `Anything`," and then they examine the program to find more precise judgements based on those that have already been made [41]. The resulting judgements are known to be true by an inductive argument over the number of judgement updates: the initial judgements are true, and each judgement derived from a true judgement is true. A benefit of such algorithms is that they may stop at any time and still have correct answers; further processing simply gives more precise answers.

All later algorithms give up this ability to stop early, in exchange for using an approach that gives more precise results. They begin with overly precise judgements such as "variable x has type `Nothing`" and then examine the program to find places where the judgement is too precise and needs to be weakened. Such algorithms must continue until they reach a fixed point and have no further weakening to perform; if they stop early then some of the types may still be too precise. This approach requires a more sophisticated argument, often based on *abstract interpretation*. Instead of inducting over judgement updates to show that the results are correct, one would typically induct over steps of execution: the results are correct in the initial state, and whenever one steps execution from one state to the next, the results remain correct.

The extra precision of such algorithms comes from avoiding self-sustaining inference loops. For example, if a program includes statements "`x := y`" and "`y := x`", then any type judged for `x` can never decrease lower than that judged for `y`, and vice versa. If either of them starts as type `Anything` then that is what they both will be when the algorithm terminates. To contrast, algorithms that start with `Nothing` must simply ensure that whenever the type of `x` increases, the type of `y` increases commensurately; `x` and `y` must have the same type, but that type can be very precise.

2.3.6 Adaptation after analysis begins

A few algorithms involve some adaptation of approach while the algorithm executes. Among these, most only adapt the approach after one complete set of judgements has been obtained; reflow analysis [60] is an example, as is Dubé and Feeley's algorithm [21].

The algorithm family of DeFouw, Grove, and Chambers [20] deserves special mention. The algorithms in this family adapt the directionality of data flow while they execute. They begin by using directional data flow, but after any one judgement has been visited more than a threshold number of times, the algorithm adapts by starting to use unification-based data flow for that judgement. Such algorithms get most of the speed benefit of purely undirected data flow, while gaining a significant amount of the benefit of directed data flow.

2.4 Scalability

Several implementations of type inference algorithms have been experimentally tested. This section gives a summary of the results of those experiments as a way to examine the scalability of existing, implemented type inferencers.

Since the experiments use different computers, code bases, and techniques of measuring performance, it is difficult to compare the results directly. Instead, this section will give three pieces of information on each experiment: the largest program on which the experimenter reported the implementation is effective, the kind of context sensitivity that the algorithm uses, and whether the algorithm uses directional data flow. The first piece of information gives an idea of how well the implementation scales, and the second two give an idea of the precision of the results of the implementation. Both directed data flow and more context sensitivity give more precise results at the expense of requiring more time.

The reported lines of code deserve some mention. The reported number below is consistently the number of lines of code processed by the algorithm. Many algorithms based on abstract-interpretation automatically ignore code that they determine to be dead code. In such cases, the amount of code analyzed might be much less than the total code in the program. This difference is important if one is considering tools for cases where the live code is a small fraction of the total code. The purpose of this section, however, is to survey the performance characteristics of existing type inferencers. For that purpose, it is appropriate to report the amount of code actually analyzed by the analyzer.

Ole Agesen performed experiments on his Cartesian Products Algorithm (CPA) in 1995 [2]. The largest example he reports is an application extraction involving the analysis of 4200 lines of live code. This example required 30 seconds of execution time on a 167 MHz UltraSparc. The analysis is context sensitive using CPA sensitivity, and it uses directional data flow.

Flanagan and Felleisen implemented a *componential* data-flow analysis and timed its execution in 1999 [25]. The largest program they analyze has 17,661 lines of code. The analysis is not context-sensitive but does use directional data flow. On a 167 MHz UltraSparc the analysis required 265 seconds.

Grove et al. implemented a variety of type-inference algorithms[2] and reported on their performance in 1997 [32]. Their results are summarized in Table 2.1. The largest dynamically typed[3] program they study is 50,000 lines of application code plus 11,000 lines of library code. They test the algorithms on a 167 MHz UltraSparc with 256 MB of memory. On the 50,000 line program, they find that none of their context sensitive algorithms complete in the available time and memory. The only context-insensitive type-inference algorithm they try (the other context-insensitive algorithms do not infer types) is based on 0-CFA [60] and succeeds on the 50,000 line program in three hours.

Grove et al. conclude from their experiments that context-sensitive algorithms such as k-**CFA** do not scale to large programs in dynamic languages such as Cecil [17]:

> 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

[2]They actually implement call graph recovery algorithms, but most of the algorithms are just as useful for type inference.

[3]The Java experiments they report are irrelevant to the present work.

	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 2.1: Each box gives the running time and the amount of heap consumed for one algorithm applied to one program. Boxes with ∞ represent attempted executions that did not complete in 24 hours on the test machine.

not scale to realistically sized programs written in a language like Cecil.

DeFouw et al. study a family of type inference algorithms that sometimes use unification-based data flow [20]. Most of them begin by using directional data flow, changing to non-directional data flow when analyzing parts of the program that are proving expensive to analyze. They seem to use the same test machine and code samples as in the Grove et al. survey of call graph recovery algorithms. They again find that purely directional analyses fail to finish in the available time for the 50,000 line program, nor even for their 20,000 line programs. Some of their hybrid algorithms do complete on the 50,000 line program, though not the hybrid algorithms that allow any context sensitivity. The fastest hybrid algorithms they tried, which have some directional data flow but no context sensitivity, finish in 50-100 seconds on the 50,000 line program.

Finally, von der Ahé implemented a type inferencer and dead code remover for Smalltalk in the Resilient environment[4] in 2004 [71], though he did not tune them for speed. His inferencer uses DCPA context sensitivity, which is more context sensitive than Agesen's CPA. He tested his implementation on a 1.7 GHz Pentium 4 Mobile CPU. He does not report lines of code analyzed or extracted, but he does report that his dead code remover succeeded in 12-14 seconds to extract a 237-method program from the 1238 methods it was embedded in.

In summary, context-insensitive analysis with undirected data flow is known to be effective on 50,000-line programs and may scale to even larger programs. Likewise, hybrid variants of such algorithms that use some directed data flow should be slower only by a constant factor [20].

The more precise context-sensitive algorithms, those algorithms that the present work focuses on, are only known at this time to scale to approximately 30,000 lines of code. Due to the cubic or slower performance of such algorithms [34], they unlikely to be practical in the near future on much larger programs, even as CPU speeds and memory sizes increase. Some modification of the existing context-sensitive algorithms is necessary to achieve scalability.

2.5 Type checking

Two other areas of related work should also be discussed: the problem of *type checking* itself, and the problem of finding more precise types in type checked languages.

The problem of type checking is to verify that a program will not commit a type error when it executes, i.e., that a program will not invoke an operation with arguments whose type is invalid [52]. Type checkers rely on having a type associated with syntactic elements such as expressions, variable declarations, and function

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

declarations. Type checking has received an extraordinary amount of attention from programming language researchers [47, 28, 46, 58, 9, 7], including the development of the Strongtalk type checker for Smalltalk [14]. Almost all type checkers rely on some amount of type inference so that programmers do not need to write down a type for every expression in a program. At the extreme are type checkers such as SML's [48] that include a type inferencer so thorough that the programmer typically needs to write down no types at all.

Type checking is a separate problem from the type-inference problem discussed in this paper. A type checker may reject a program outright, while the type inferencers studied in the present work must succeed on any program. Programmers using a type checker typically expect to modify their program in response to issues identified by a type checker. To contrast, programmers using a type inferencer (or a tool based on type inference) are seeking to find more information about an existing program, and they will not necessarily change the program even if the tool points out potential problems.

This difference results from a fundamental difference in the property proved by each tool. A type inferencer must only find types that are *correct*, that is, large enough to include all values that the associated syntactic element will hold when the program runs. A type checker must find types that are additionally small enough that any operation the program applies to the associated syntactic element is appropriate to the type. Since some programs do have type errors, it is inevitable that a type checker must reject some programs. A type inferencer, meanwhile, can succeed on any program; at worst it can assign a type of `Anything` to everything in the program. In the extreme, if a type inferencer analyzes a program that is certain to commit a type error when it runs, the inferencer must still be careful to find correct types for the portion of execution preceding the type error.

Another separate problem is that of improving the types that a type checker finds. For example, given a variable in Java[30] that has an abstract Java interface type, one might wish to learn more specifically which concrete classes the variable will actually hold at runtime. In many cases it will not hold every possible class that matches the interface, and in some it will hold only one class. Examples efforts are those of Tip and Palsberg in 2000 [69], and Wang and Smith in 2001 [72]. Since such algorithms start with the reasonable types and call graphs given by the language, they solve an easier problem than the present one.

2.6 Knowledge-based systems

Knowledge-based systems, also called *expert systems*, provide a general theory for the present area of enquiry. A knowledge-based system has an architecture with four components: a *knowledge-acquisition module*, a *knowledge base*, an *input/output interface*, and an *inference engine* [49]. A demand-driven type-inference algorithm follows this architecture.

The acquisition module of a knowledge-based system provides the initial information and inference rules that the system may use. For a type inferencer, the acquisition module includes two parts. First, it includes information about the particular program being analyzed. Such information is provided through tools such as parsers and static semantic analyzers. Second, it includes inference rules particular to the type-inference algorithm. The acquisition-level information and rules used by **DDP** are described in Chapter 5 and Chapter 7 respectively.

The knowledge base holds the information from the acquisition module as well as information inferred as the analyzer runs. This information can include control information such as what goals the inferencer is currently pursuing. For a type inferencer, the knowledge base includes type judgements and other control- and data-flow judgements that have been inferred about the program. The judgements **DDP** uses are described in Chapter 6.

The input/output interface interacts with the user. Most type inferencers use a simple interface that simply accepts questions from a user and then reports results. In general, however, an input/output interface might interact with a user as it deduces information and might expend considerable sophistication on the problem of explaining inferred results. Mr. Spidey is just such a tool with a sophisticated interface [24]. The input/output interface for **DDP** is the Chuck program browser described in Chapter 9.

The inference engine repeatedly applies rules of inference to update the knowledge base. Typical type inferencers use a simple inference engine that simply applies every available inference rule until there are no more possible updates to the knowledge base. Adaptive demand-driven algorithms, discussed above, are an exception: such algorithms have a variety of available strategies and choose among those strategies in some fashion. The most interesting part of **DDP** is its adaptive inference engine, described in Chapter 3 and section 4.8.

2.7 Semantics of Smalltalk

The formal work in this paper is based on a new description of Smalltalk's semantics that is detailed in Chapter 5. It is worth reviewing a few existing descriptions and the new one's relation to them.

The earliest full description of Smalltalk semantics appears in *Smalltalk-80: The Language and Its Implementation*, by Goldberg and Robson [29], often referred to as *the blue book*. In addition to a lengthy informal description and rationale, the blue book includes a complete interpreter written in the language itself. Most early semantics of Smalltalk refer to the blue book's definition of the language.

Unfortunately, the blue book's description does not give blocks the full semantics of closures. It defines blocks without temporary variables at all. Later implementations of Smalltalk include full closure semantics including reentrant blocks and nested mutable variables. However, all semantics that mimic the definition of Smalltalk in this book must necessarily use a limited definition of blocks.

Nested mutable variables are a ubiquitous feature of modern Smalltalk implementations, and accordingly they are required by the current ANSI Smalltalk standard [8]. Unfortunately, they add complexity to descriptions of the semantics and non-trivial requirements for correct program analysis. Given these factors, the need to describe nested mutable variables is the most compelling reason that a new semantics of Smalltalk is included in the present work.

Wolczko has developed a denotational semantics of Smalltalk [73] as part of a larger project studying object-oriented semantics in general [74]. His Smalltalk semantics describes a variety of language features including not only the expected features such as objects, classes, messages and methods, but also primitives (including three important examples) and arrays. Nevertheless, in order to stay true to the blue book's semantics, Wolczko begrudgingly omits nested mutable variables from his Smalltalk semantics. His paper describing Smalltalk semantics includes a number of comments on the lack of nested mutable variables and other limitations of blocks from the blue-book specification. For example, Wolczko writes:

> The absence of temporary variables from blocks was a curious omission in the design of Smalltalk. Later we shall meet other strange features of blocks. [73]

Wolczko's Smalltalk semantics consistently avoids a general description of nested temporary variables. Instead, he suggests treating nested temporary variables as syntactic sugar, a language feature that is unimportant semantically and can be interpreted by rewriting all uses into features that do exist in the low-level semantics. Wolczko describes two techniques for rewriting Smalltalk blocks that access non-local variables: fixing the values of non-local accesses at the time a block is evaluated into a closure, and replacing mutable variables by non-mutable variables that hold a reference to a mutable cell of memory. The combination of these rewrites are sufficient to capture the semantics accurately, albeit indirectly. This rewriting approach is a good trade off for a project whose purpose is to focus on the specifically object-oriented parts of the language semantics.

The present work has a different purpose: studying data flow in Smalltalk. Since assignments to temporary variables are a common and tricky mechanism for data flow, it is imperative to describe nested mutable variables at some level in the associated theory. The present work elects to describe nested mutable variables directly at the level of the semantics. This approach requires a somewhat more complex description of the semantics, but in return, it removes the need to add additional lemmas and mathematical structures at a higher level to accurately describe data flow through nested temporary variables. Further, it results in a simpler correctness theorem whose statement is closer to the language semantics. Additionally, some language features, including arrays and most primitives, have straightforward effects on data flow analysis

a, b	$::=$		terms
		x	variable
	$\mid$	$[l_0 = \varsigma(x_0)b_0, \ldots, l_n = \varsigma(x_n)b_n]$	object formation
	$\mid$	$a.l$	field selection or method invocation
	$\mid$	$a.l \Leftarrow \varsigma(x)b$	update of field or method

Table 2.2: Core of Abadi and Cardelli's theory of objects

(the present work conservatively analyzes flow through arrays), and a new semantics is an opportunity to remove those features that, for our purposes, provide more of a distraction than an elucidation.[5]

Abadi and Cardelli, too, have developed a general theory of object-oriented semantics [1]. Their theory is tuned for discussion of static type systems for object-oriented languages. They discuss a number of static-type issues such as subclassing versus subtyping, types for class-based versus object-based languages, self types, universally and existentially quantified types, and covariant typing. As with Wolczko's semantics, Abadi and Cardelli's choices are appropriate for their purpose but cause difficulties for developing the theory behind a data-flow algorithm. The syntax of Abadi and Cardelli's core language is given in Table 2.2. Notice that methods and fields are treated equivalently. The language thereby allows copying of methods from one object to another, a powerful feature normally reserved in a language for reflective development tools. On the other hand, higher-level constructs such as classes, inheritance, and blocks (lambda abstractions), are left out of the core language and left to be treated as syntactic sugar. These choices work well for Abadi and Cardelli's expressed purpose of studying object-oriented semantics and the associated static type systems. However, for the present purpose, the theory is simplified if extremely powerful features like method update are removed while higher-level features important to analysis are described directly.

[5]Of course, the implementation must correctly support these features even though the theory ignores them. The details are given in **??**.

Chapter 3

Developing a new algorithm

The problem concerning the present work is to infer types in large programs, particularly as an aid to program-understanding tools. Given the existing work on the problem, how should one proceed? This chapter develops a new type inference algorithm to address this problem. The algorithm is not yet described in full. A complete but informal description is in Chapter 4. A formal description of the analysis rules, the trickiest part of the algorithm, is developed in Chapter 5 through Chapter 7.

3.1 Observations

Consider a few observations from the existing published work and on the nature of the problem itself. These observations point the way forward to an algorithm more likely to solve the stated problem.

First, observe that existing context-sensitive algorithms do not scale to larger programs. Even 0-CFA has difficulty with 50,000-line programs [32]. CPA and the k-CFA's become impractical at even smaller sizes. If one wants to analyze programs with hundreds of thousands of lines of code, then one should seek some fundamental change from the existing published algorithms.

Second, note that within any realistic large program, there are many type inference questions that are easy to answer. If nothing else, the types of literal expressions are easy to derive. For example, the type of `42` is clearly something like `Integer`—it does not matter where the `42` is embedded in some large program. Additionally, realistic programs tend to have many variables where some short investigation can find a type. For example, if a variable `Pi` is only assigned one value in the program, and that value is a literal, then the type of `Pi` is the type of the literal. If one wants a useful algorithm, then one should seek an algorithm that can at least find answers to the easy questions.

Likewise, in most realistic large programs, there are type inference questions that are impractical to answer. Consider the argument of a method named `#new` :. There are many hundreds of expressions that send the message `#new` :, and deciding the type of the argument to the method requires coping with all of those expressions in some fashion. For at least some `#new` : methods, this is likely to be impractical in a sufficiently large program. Therefore, if one wants a scalable algorithm, one should seek an algorithm that can give up at some point instead of tilting at every windmill indefinitely.

Finally, there are precise type inferences that do not require precise types at every step of the derivation leading to the final inference. For example, consider an expression like "`regex matches:  someString`". To find the type of the expression, the inferencer will find a type for `regex` and then analyze each method that, based on that type, might be invoked by the statement. However, it might not matter whether `regex` is determined to be precisely the set of regular expression classes, or the ultimately imprecise `Anything` type; in either case, the inferencer will find that all `matches:` methods may be invoked, and thus it will find the same type for the expression "`regex matches:  someString`". Because of such scenarios, a type inference algorithm can give up on subproblems without necessarily losing precision in the final answer. If giving up appears to be necessary, the inferencer should at least attempt to give up on subproblems before giving up on the main problem posed to the inferencer.

3.2 Approach

The previous observations lead to several ideas for building a scalable and useful algorithm.

One general idea is that the algorithm could spend some resources searching for an answer and then give a trivially correct answer if none can be found before the allocated resources are exhausted. This general approach implies that easy questions will be answered well, while difficult questions will be answered poorly but in reasonable time.

For this approach to be effective, it should be possible to use a different strategy on each questions that has been posed; otherwise, if any one question is difficult, the algorithm would be forced to give up on the entire program. A *demand-driven* algorithm has the necessary property. A demand-driven algorithm answers each question individually, thus gaining has the flexibility to choose a different strategy for each question.

A natural refinement is to allow the algorithm to give up on individual questions instead of just on the initial posted goal. This way, the algorithm can give precise types to an additional number of queries: those queries that have expensive subgoals that do not influence the final result. This refinement is called *pruning subgoals*. A goal is pruned by giving it a trivially correct answer, thus ensuring that the goal needs no subgoals.

In order to support subgoal pruning, the goals of the demand-driven algorithm must be formulated carefully. For a goal to be prunable, it must admit some answer that is definitely true, and that answer must be quickly computable—ideally, in constant time. For example, the goal "what is the type of `x`?" is prunable, because one can answer "x is of type `Anything`." On the other hand, one cannot prune the goal "summarize the effects of calling method `m`, and update all goals to account for those effects".

This approach could be summarized by framing the problem as a knowledge-based system (KBS) [49] and then using a non-trivial inference engine. The propositions the KBS processes are data-flow judgements; the goals of the KBS are the same as the goals of this approach; the inference rules of the KBS are justification tactics; and the non-trivial inference engine continually chooses for each goal whether that goal should be pruned or pursued further.

3.3 Structure of the DDP algorithm

The **DDP** algorithm uses the approach described previously. It is demand-driven, and it prunes subgoals. This section gives the overall structure of **DDP**. Later chapters elaborate on several details.

The overall algorithm, summarized in Figure 3.1, is a standard demand-driven algorithm modified to sometimes prune goals. A *goal* is a question the algorithm is trying to answer. Every goal being pursued by the algorithm has a tentative answer to its question. As the algorithm progresses, those answers are repeatedly adjusted.

The standard part of the algorithm is that there is a set **worklist** holding a set of goals that need to be updated. The algorithm repeatedly removes a goal from **worklist** and updates its answer. If the answer actually changes, then any goals depending on the updated goal are added back to **worklist** for future consideration—this way, tentative answers to goals can be updated in light of new information whenever the subgoals they depend on are given new answers. The algorithm terminates when **worklist** is empty and thus all relevant goals are consistent with their subgoals. At that point, all relevant goals have answers that are in fact correct.

The **UpdateOneGoal** function modifies the current answer of one goal to be consistent with the answers to the goal's subgoals. For example, it might change the answer of the goal from "x is an Integer" to "x is an Integer or a Float", in order to account for new information in the goal's subgoals. The function **Update** performs this modification, and it returns a boolean indicating whether the goal's answer needed changing (it is possible that the update leaves a goal with the same answer as before).

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

  while worklist ≠ ∅ do
    if pruner wants to run
    then Prune()
    else UpdateOneGoal()

  return GoalAnswer(rootgoal)


procedure UpdateOneGoal()
  Remove g from worklist
  changed := Update(g)

  if changed then
    deps := GoalsNeeding(g)
    worklist := worklist ∪ deps


procedure Prune()
  for g ∈ ChoosePrunings() do
    prune g
  worklist := Relevant(rootgoal)
```

Figure 3.1: The **DDP** algorithm.

The precise behavior of **Update** is given in Chapter 7. Note, though, that if the goal being updated needs new subgoals that do not already exist, then those goals are created, given a maximally precise answer (e.g., the empty type holding no values), and added to **worklist**. If an update causes a change to the goal's answer, then **UpdateOneGoal** adds all goals that depend on the goal to **worklist**.

The modification from the standard demand-driven algorithm is that, in some iterations, the algorithm calls **Prune** and *prunes* goals instead of updating a goal. The **ChoosePrunings** function chooses which goals should be pruned and is described further below. **ChoosePrunings** is a heuristic, and there are many possibilities for its specific behavior; see section 4.8 for some of them. Whatever goals the function chooses are pruned by being given trivially correct answers, thus ensuring that they require no subgoals. After the chosen goals are pruned, **worklist** is reset to hold precisely **rootgoal** plus all direct and indirect subgoals of **rootgoal**.

To increase the effectiveness of pruning, the **GoalsNeeding** function should not return goals that have become irrelevant due to pruning; otherwise, some pruning would essentially be undone. In order to efficiently return this limited set, an extra set **completed** can be maintained. The **completed** set holds those goals that have been updated and whose immediate subgoals have not had a change in value. Whenever a goal is updated, it should be added to **completed**, and whenever a goal is added to **worklist**, it should be removed from **completed**. Thus, as the algorithm progresses, goals that are relevant move back and forth between **completed** and **worklist**, always being in at least one of them. Goals that are irrelevant due to pruning are removed from both sets. The **GoalsNeeding** function can then return only goals which are present in either **completed** or **worklist**.

3.4 An example execution

This section traces one execution of **DDP**, in order to clarify how the general algorithm works. The example execution analyzes a program that includes the code of Figure 3.2, among a great deal of other code that is not listed.

The figures showing the progress of the algorithm show the *knowledge base* of all relevant goals. An example goal is shown in Figure 3.3. On the left is the question the goal attempts to answer: *What is X?* On the right is a tentative answer to that question: the type *Bottom*, a type designating that `X` is never assigned a value. The center box inside the goal is empty. If it instead were marked with a `J` in the middle, then that goal would be justified with respect to its immediate subgoals. Since it does not have a `J`, this goal needs more work.

The worklist of **DDP** is not shown explicitly in the diagrams. Instead, the worklist consists of those goals that are not justified and thus are not marked with a `J`. To simplify the figures, the ordering of the worklist is not shown; it is irrelevant for the important aspects of **DDP**.

The algorithm begins as **InferType** is called with argument `X`. The algorithm inserts a type goal for `X` into the knowledge base, arriving at Figure 3.3. Notice that whenever a new goal is created, it is given an initial answer that is extremely precise, e.g., type $\perp$, an answer which is almost certainly overly specific. That answer will be broadened as the algorithm progresses.

Suppose that the algorithm proceeds to call `UpdateOneGoal` instead of `Prune` for several iterations. The first goal it chooses to update must be the one for the type of `X`. `Update` in this case will find all statements that modify `X`. In this case there are two, and the algorithm creates a subgoal for each one of them. Then, the algorithm updates the type of `X` to account for the current answers in the subgoals; since the subgoals are newly created, they still have answers of `Bottom`, and thus the type of `X` is also left as `Bottom`. The root goal is marked as justified, arriving at Figure 3.4.

The algorithm must now choose a new goal to update. It can choose either `Y` or `p1`. Suppose it chooses `Y`. There is only one statement in the program that modifies `Y`, and it assigns the literal `10` to the variable. Thus, the type of `Y` is precisely `Integer`.[1] The algorithm updates the answer to `Y`'s type goal. Since this type is a change from the old type, i.e. `Bottom`, the algorithm also marks as unjustified all goals that depend on this

[1]For simplicity, this example ignores the fact that in Smalltalk all variable bindings hold `nil` when they come into existence.

- Class A, method foo:, is:

```
foo: p1
  X ← Y.
  X doStuff.
  X ← p1.
  X doMoreStuff.
  ^X
```

- Class A, method bar, is:

```
bar
  Y ← 10.
  ^Y
```

- Class A, method baz, is:

```
baz
  | s |
  s ← self.
  s foo: Y.
  ^s
```

- Class A, method extraneous, is:

```
extraneous
  Q ← 10.
  Q foo: Q.
  ^Q
```

Figure 3.2: Code for example execution. The relevant methods of some large program are listed; it is assumed that a great deal of other code is also present in the program. The syntax is Smalltalk; the details are described in Chapter 5. In agreement with the Smalltalk convention, all uppercase variable names refer to global variables.

type goal (i.e., it puts those goals back on the worklist). In this case, the only goal depending on Y's type goal is the root goal. The execution state arrives at Figure 3.5.

The algorithm might now choose to update the root goal. The same analysis is repeated from the first time this goal was updated, but now instead of creating new goals for Y and p1, the algorithm can reuse the goals that already exist. The algorithm updates the answer to the root goal to be "Integer or Bottom," which, since Bottom is empty, is the same as simply "Integer". The root goal is then marked as justified. Since there are no goals depending on the root goal in this example, there are no other goals to mark as unjustified. The execution state is summarized in Figure 3.6.

The algorithm might now update the goal for p1. To find the type of p1, which is a parameter, the algorithm first tries to find all message-send statements that might invoke the parameter's method, in this case the #foo : method of class A. A new goal is created to try and find these send statements. Initially, the goal's answer is that no send statements in the program can possibly invoke the method, and thus Update concludes that p1 is never assigned a value. The execution state is now described in Figure 3.7.

Suppose now that the algorithm decides to prune some goals; i.e., it calls Prune instead of UpdateOneGoal. Prune must choose some goals to prune. Suppose it chooses the newly created senders goal for A's #foo : method. The goal is given a trivially correct, extremely conservative answer, and all goals depending on the goal are marked as unjustified. After the pruning finishes, the algorithm removes from consideration any goals no longer relevant to the root goal; in this case, however, all goals are still relevant, because the single goal that was pruned had no subgoals. The execution state is now described in Figure 3.8.

There is only one goal to update, so the algorithm must choose to update the type goal for p1. That goal must now account for all statements that might invoke A's #foo : method. Human analysis can show that one of the two statements actually calls some other #foo : method, but the senders goal was pruned and thus does not have as precise of an answer as a human can find. The type goal for p1 must then consider type goals for both Y and Q. A type goal for Y already exists and is reused, while a new goal must be created for Q. The execution state reaches that described in Figure 3.9.

The algorithm might then update X, causing no change except to mark X's goal as justified, reaching the state in Figure 3.10. The algorithm might then update Q (Figure 3.11) and then update p1 again (Figure 3.12). Since the type of p1 did not change this time, there is no need to unmark any goals as justified. Since no more goals need justification, the algorithm terminates.

At this point, all goals within the knowledge base are justified with respect to each other. The justification rules are such that all goals must, in this circumstance, have correct answers. Thus the root goal's answer is correct, and a correct type for X is Integer.

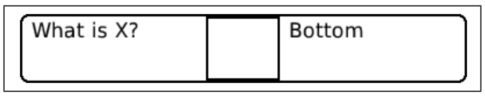


Figure 3.3: Example: The initial state of the knowledge base. There is one question, “What is X?”, and it has a tentative answer, Bottom.

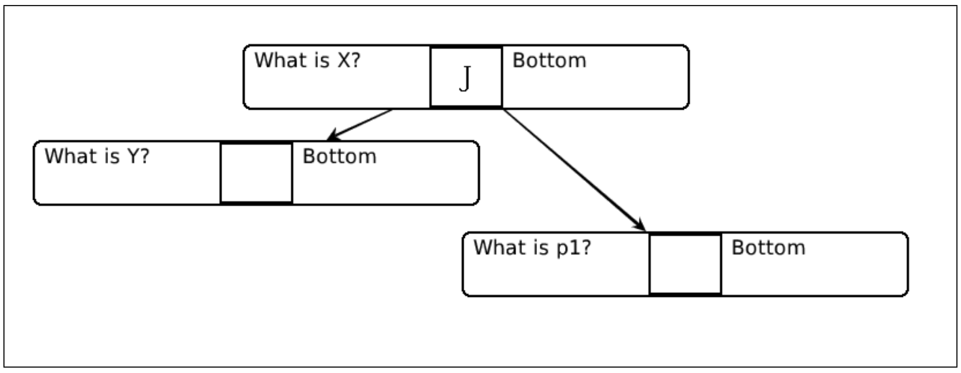


Figure 3.4: Example: The root goal is updated. It now has two subgoals. Since the root goal’s answer is consistent with all of the goal’s subgoals, the goal is marked as justified.

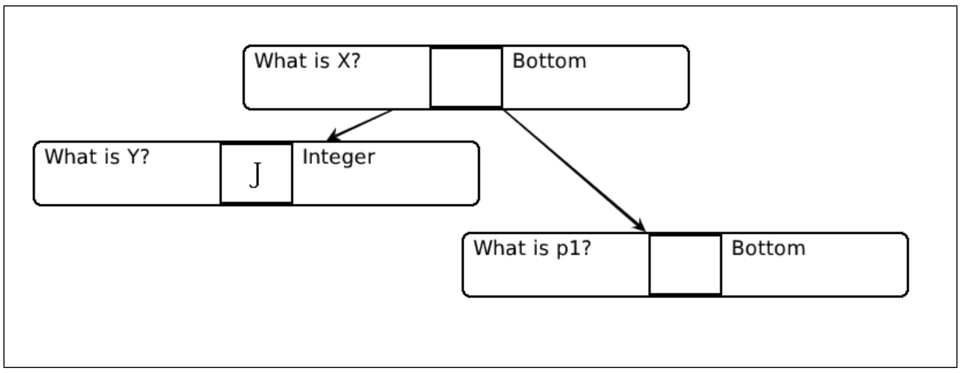


Figure 3.5: Example: The type goal for Y is updated. Since the root goal depends on the type goal for Y, the root goal is no longer justified.

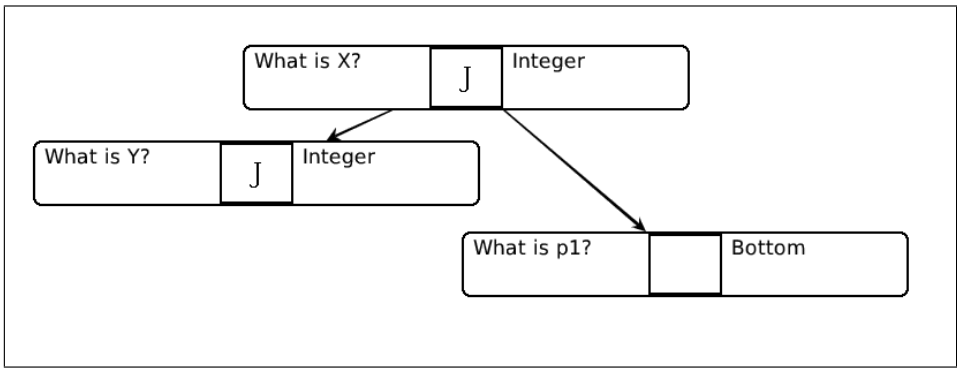


Figure 3.6: Example: The root goal is updated again. It is now consistent with its subgoals, and so it is marked again as justified.

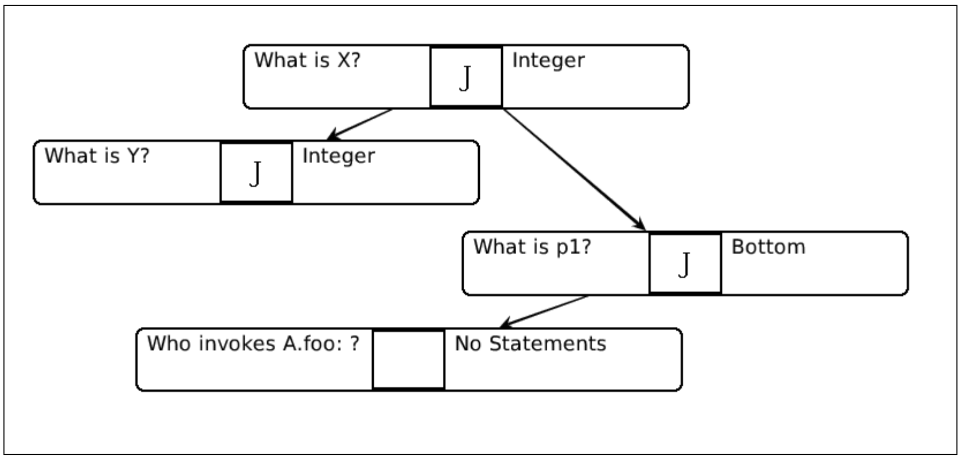


Figure 3.7: Example: The goal for `p1` is updated. Since `p1` is a parameter of method `A.foo:`, the algorithm must find the senders of `A.foo:` in order to find the type of `p1`.

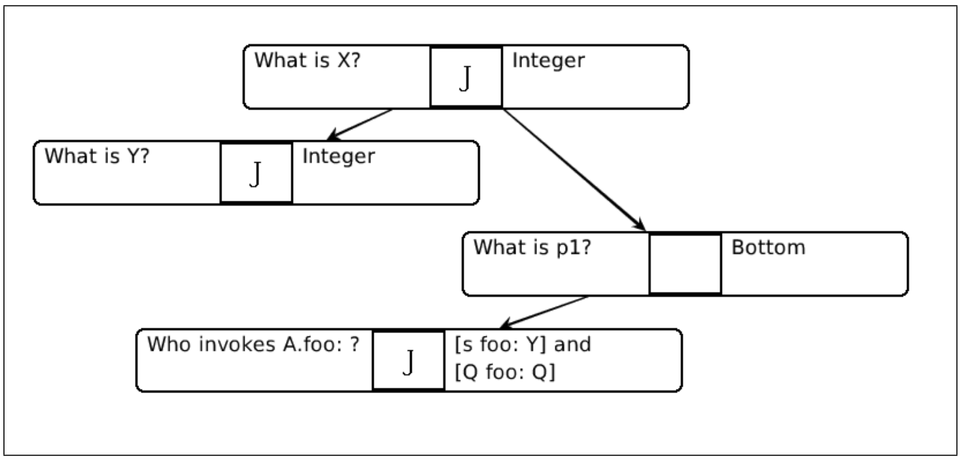


Figure 3.8: Example: The senders goal is pruned. The goal now has a sufficiently conservative answer that no subgoals are required.

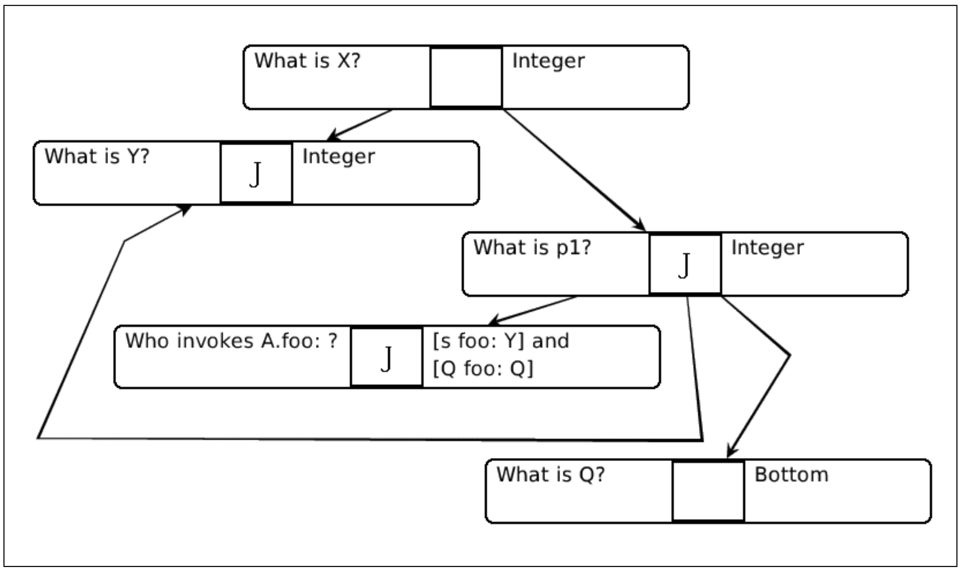


Figure 3.9: Example: The goal for `p1` is updated again. Two new subgoals are required, and the root goal is no longer justified. Notice that the existing goal for `Y` is reused.

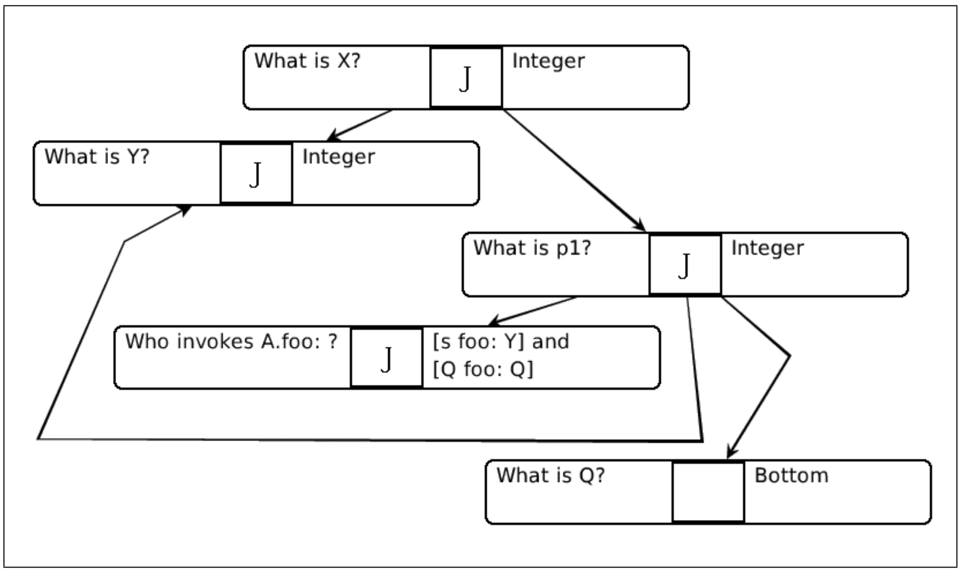


Figure 3.10: Example: The goal for X is revisited. Its answer needs no change.

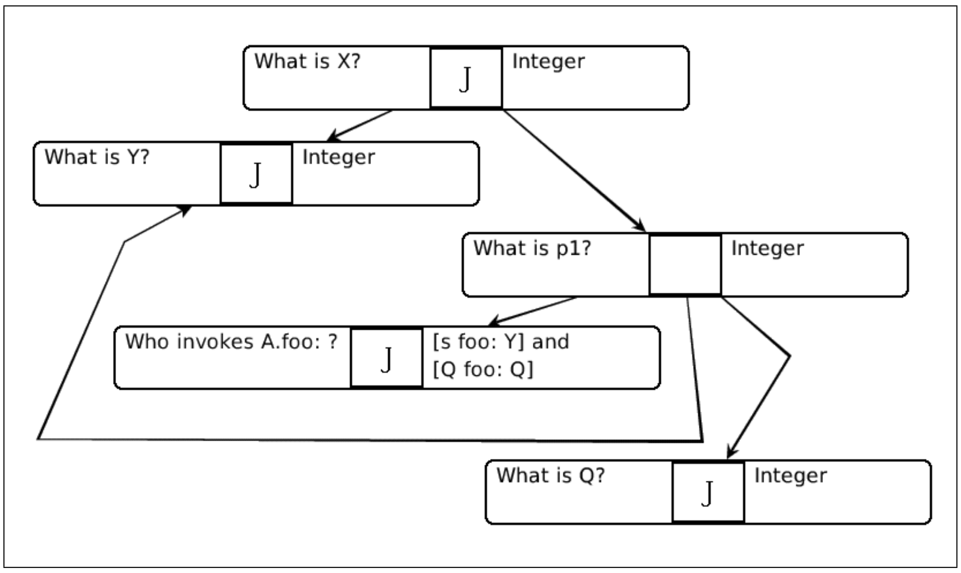


Figure 3.11: Example: The type goal for `Q` is updated.

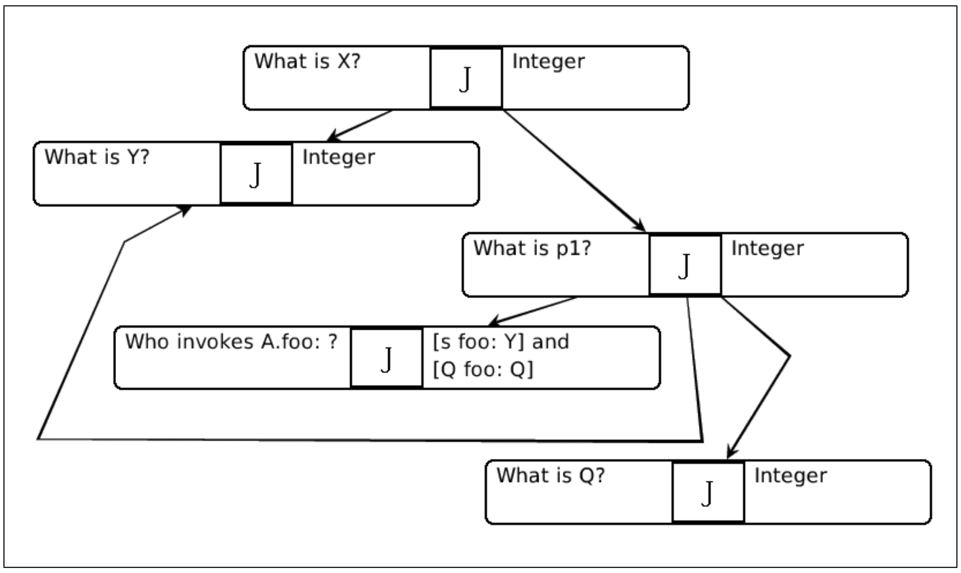


Figure 3.12: Example: The goal for `p1` is updated again. All goals are now justified, so the algorithm terminates.

3.5 Properties of the general algorithm

The **DDP** algorithm has several nice properties. First, the time of execution appears to depend mostly on the number of goals analyzed and the number of times they are updated. It does not appear to depend much on the size of the program. Therefore, assuming these intuitions are correct, the algorithm should complete quickly whenever the number of nodes is restricted, even if the analyzed program is large.

Second, the algorithm finds many short type derivations where possible. One-step derivations, such as the type of a literal expression, are clearly found by **DDP**. Additionally, if a short multiple-step derivation happens to fit within the goals that are not pruned, then the **DDP** will find that multiple-step derivation as well. Note that this includes multiple-step derivations for which some of the subsidiary judgements are not precise. Overall, there are several cases where **DDP** finds a precise answer to a type query, and when it cannot find a precise answer, it will give up in reasonable time.

Finally, **DDP** can be tuned to use more or less effort. By pruning more severely, the algorithm should finish more quickly. By pruning less severely, the algorithm should finish with better results. Thus, the severity of pruning provides a knob on the algorithm which trades speed for precision.

Chapter 4

The DDP algorithm

4.1 Overview

This chapter expands on Chapter 3 to give an informal description of the entire **DDP** algorithm. Some technical detail is omitted in the interest of readability. Nonetheless, enough information is provided both to implement **DDP** in Smalltalk or to form a basis for adapting it to another language. Knowledge of Smalltalk is not assumed, but a general understanding of object-oriented programming is probably required.

The algorithm is presented as a base algorithm followed by two refinements. The base algorithm analyzes a core object-oriented language. The two refinements support additional Smalltalk features that are used widely in practice: blocks and the `perform` family of methods.

4.2 The base language analyzed

This section describes ST_0, the core language analyzed. It is essentially Smalltalk, but it has some simplifications that fall into two categories. First, some elements of the full language are not described because they are straightforward to analyze but complicated to describe. Examples are literals in their full generality, the various forms of singleton variables other than global variables (class variables and pool variables), and the full list of primitive methods.

The second simplifications are that blocks and the `perform` methods are not described until later in this chapter. These features add significant complexity to the analysis. By deferring their discussion, the description of the core algorithm becomes simpler. Additionally, these features are not available in many object-oriented languages, and a reader who is contemplating adapting **DDP** to some other language can safely focus on those extensions that are present in that language.

The concrete syntax is ALGOL-like, even though the semantics follows Smalltalk, in order to remain accessible to a wide audience. Smalltalk enthusiasts can hopefully forgive the author for writing in the vulgar.

For readability, code is bracketed with $[\![ ]\!]$ delimiters. There is no semantic significance to these brackets.

The language is object-oriented with single inheritance. All values are objects, including primitive values such integers and textual characters. There is no distinction between "public" and "private" methods and variables. Instead, all methods are invocable by all objects, and all instance variables are only accessible in the methods of the class they are defined in.

The syntax is summarized in Figure 4.1. A program consists of global variables, classes, and methods. Each class has a single superclass (unless the class is the root class Object), a list of instance variables, and a list of methods. Classes in a valid program must have a superclass hierarchy that is a proper tree rooted at a class named Object.

```
   ⟨program⟩  ::=  (⟨global⟩ | ⟨class⟩) ∗
    ⟨global⟩  ::=  “global” ⟨identifier⟩
     ⟨class⟩  ::=  “class” ⟨identifier⟩ [: ⟨identifier⟩] “{” ⟨method⟩ ∗  “}”
    ⟨method⟩  ::=  “method” ⟨identifier⟩ “[“ “primitive” ⟨identifier⟩ “]” ⟨block⟩
     ⟨block⟩  ::=  “{” ⟨identlist⟩ ⟨identlist⟩ ⟨expseq⟩ “}”
    ⟨expseq⟩  ::=  (⟨expression⟩ “; ”) +
⟨expression⟩  ::=  ⟨literal⟩
               |   ⟨identifier⟩
               |   “new” ⟨identifier⟩
               |   ⟨expression⟩ “.” ⟨identifier⟩ ⟨explist⟩
               |   ⟨expression⟩ ⟨symbol⟩ ⟨expression⟩
               |   “return” ⟨expression⟩
               |   “(” ⟨expression⟩ “)”
   ⟨explist⟩  ::=  “(“ [ ⟨expression⟩ (“, ” ⟨expression⟩)∗ ] “)”
 ⟨identlist⟩  ::=  “(“ [ ⟨identifier⟩ (“, ” ⟨identifier⟩)∗ ] “)”
⟨identifier⟩  ::=  “x”, “y”, “+”, “*”, etc.
    ⟨symbol⟩  ::=  “+”, “*”, etc.
   ⟨literal⟩  ::=  “123”, “'hello'”, “5.16”, etc.
```

Figure 4.1: Syntax of ST$_0$. The extended BNF uses () for grouping, [] for optional items, * to indicate zero or more repetitions, and + to indicate one or more repetitions.

Each method has a name, called its *selector*, an optional primitive tag, and a *block* that defines the behavior of the method when it runs. All methods in a class must have distinct selectors. Primitive methods are used to perform functions outside the core semantics, such as arithmetic and input-output, as well as a few language features such as blocks and `perform`. Primitives routines are referenced by an identifier such as "+" or "`print`," and must be defined within the virtual machine.

A *block* represents behavior in ST_0 and its derivatives. It includes a list of formal parameters, a list of local variables, and a list of expressions that should be executed whenever the block is evaluated. The formal parameters must be specified whenever the block is evaluated. The local variables are initialized to the special value `nil` when the block begins execution, and afterwards the expressions in the block can access and redefine those local variables arbitrarily. Expressions are as follows:

- $[\![lit]\!]$. Inject a literal into the computation. This chapter does not give a full syntax for literals, but typical examples of literals would be the integer 1234 and the string `'hello'`.
- $[\![var]\!]$. Read from a variable that is in scope. ST_0 is lexically bound, so `v` must be a parameter or local variable of the block the expression is in, an instance variable of the class the expression is in, or a global variable.
- $[\![$`new` *identifier*$]\!]$. Instantiate the class named *identifier*, yielding a new object.
- $[\![var := exp]\!]$. Evaluate expression *exp* and assign it to the variable *var*. The same notes apply regarding lexical binding, but additionally *var* may not be a parameter.
- $[\![exp_0.sel(exp_1 \ldots exp_m)]\!]$, $[\![exp_0 \;\; sym \;\; exp_1]\!]$. Send a message to an object, thus invoking a method. The first form of the syntax is more general and allows invoking a method of any name and any number of arguments. It evaluates the exp_i's from left to right and then invokes the method named *sel* in the object to which exp_0 evaluated, specifying the objects from exp_1 through exp_m as actual parameters. If no method is available named *sel*, then the program halts. The second syntax has the same semantics but is limited to binary symbols such as + and *.
- $[\![$`return` *exp*$]\!]$. Evaluate *exp* and return the resulting value from the currently executing method.

The most interesting feature of ST_0 is its object-oriented message sending. Since message sending adds considerable complexity to the analysis, it is worth dwelling on its terminology and precise semantics. The participants of a message send are as follows:

- A *message send* is a request to invoke a method on some object. Message sends result from evaluating message-send expressions such as $[\![$`3 + 4`$]\!]$.
- A *method* is a named body of code in some class. For example, if one evaluates $[\![$`3 + 4`$]\!]$, then the + method in class SmallInteger will respond.
- A *receiver* is an object that is being sent a message. In the expression $[\![$`3 + 4`$]\!]$, the receiver is the number `3`.
- A *selector* is the name of a method. In the expression $[\![$`3 + 4`$]\!]$, the selector is the identifier +.
- A *message* is the combination of a selector with a complete set of arguments. In the expression $[\![$`3 + 4`$]\!]$, the message is "`+ 4`".

The semantics of a message-send are as follows. First, the responding method is located by searching the receiver's class for a method with the specified selector. If the receiver's class has no matching method, then its superclass is searched, followed by its superclass's superclass, and so on up the inheritance chain until reaching class Object. If no class in the inheritance chain has a method with the specified selector, then the message send fails and the program halts. Note that the responding method depends on the class of the receiver. Message-sends are thus polymorphic: the same expression can invoke a different method each time it is evaluated during program execution.

Once the responding method has been located, a new environment is created mapping the formal parameters of the method to the the actual arguments specified in the message send. The expressions of the method then execute in the newly created environment. If the method executes a `return` expression, then the method ceases executing and the message-send expression itself evaluates to the returned value. If a method's expressions all evaluate but no `return` expression executed, then the program halts; it is illegal to end a method without returning some value.

If the responding method is a primitive method then the semantics are slightly different. First, the appropriate handler for the named primitive is executed . If the handler succeeds, then the handler chooses the return value and the regular expressions in the method are ignored. If the handler aborts for any reason, then the regular expressions of the method execute just as if the method did not have a `primitive` designation. The expressions in a primitive method are thus called *fail code*, because they execute only when the primitive fails.

A complete set of primitives are not be given in this chapter, but the following primitives will be used in the course of discussion:

- `print`. This primitive expects no arguments. The receiver should be a string, and that string will be printed out to the user. It returns the receiver.

- `+`. This primitive expects one argument. It adds the receiver to the argument and returns the resulting value.

An extra primitive will be added with each of the extensions (blocks and `perform`) described below.

A few technical details are glossed over in this chapter. First, it is assumed that all variables have unique names. Thus it is meaningful, for example, to discuss "the method of which *var* is a parameter," because it is not allowed to have parameters of different methods with the same name. Implementers must be careful to use some sort of fully qualified variable names to obtain this uniqueness. Additionally, it is assumed that all expressions are labeled, so that all references to an expression refer not to the expression in general, but to a specific occurrence of that expression somewhere in the program. The labels are consistently ignored in the text of this chapter, but an implementer must be careful to use labeled expressions whenever it is necessary to distinguish different occurrences of the same expression.

The mathematical work in subsequent chapters is more careful with these technical details.

4.3 DDP goals

Recall that **DDP** is *demand-driven*. The algorithm progresses by posing questions to itself and then finding and improving answers to those questions. **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 method.

The queries of **DDP** can be described on two axes: backward versus forward flow, and data- versus control- flow. As Figure 4.2 shows, all four possibilities in the cross product are used by **DDP**useful.

Figure 4.2 also shows the dependencies between goals and the subgoals they use to find their answers. For example:

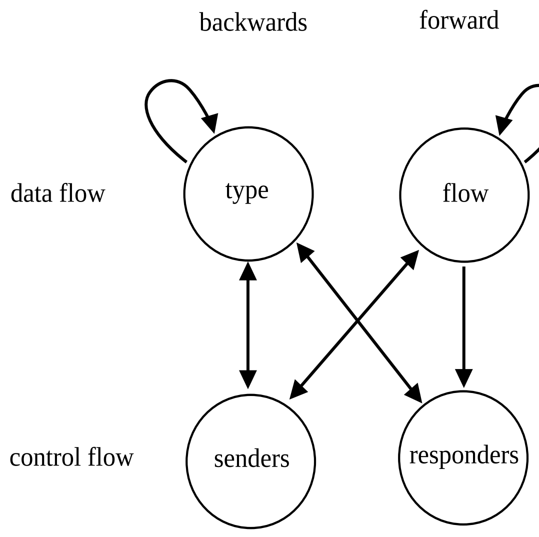


Figure 4.2: The four queries Chuck can answer along with their dependencies on each other.

- 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 [59].

4.3.1 Flow queries

A *flow query* is written $f \rightarrow^* ?$ and asks where the value produced by some variable or expression f will flow when the program runs.

The answer to a flow query is a *flow position*. 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.
- Sets of the above. Any finite set of simple flow positions is itself a flow position.

These flow positions are additionally discriminated by static contexts as described below in section 4.4.

For efficiency reasons, the value $\top_{fp}$ holding all possible flow positions is implemented as a special case requiring only constant space to represent and constant time to process.

4.3.2 Type queries

A *type query* asks what kind of values a variable or expression will hold when the program runs. The answer to a type query is a *type*. In the base algorithm there are only two kinds of types processed by **DDP**:

- Individual classes. For example, `PlayingCardDeck` is a valid type which includes all instances of class `PlayingCardDeck`.
- Sets of the above. A set of simple types is also a type. The answer to a type query is typically a set and not an individual program element.

Like flow positions, type queries are additionally discriminated by context as described below.

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

4.3.3 Responders queries

A *responders query*, denoted $expr \star b \xrightarrow{send} ?$, asks what methods might respond when a particular message-send expression executes. Unlike the stock Smalltalk program browser, a **DDP** responders query can use type queries to gain a more precise answer. **DDP** does not have to answer a responders query with every method that has the same name that *expr* specifies, but can instead answer the subset that is consistent with inferred type information.

4.3.4 Senders queries

A *senders query*, denoted $meth \xleftarrow{send} ?$, asks what expressions might invoke a specified method. As with responders queries, **DDP** can use type information to provide more precise information than is provided by the standard Smalltalk program browser. Instead of returning all message-send expressions which send the same selector as a queried method's name, **DDP** can return the subset that is consistent with type information.

4.4 Context

As discussed in subsection 2.3.2, **DDP** uses **CPA**-style contexts throughout its data-flow judgements. Such context is in fact 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.
- *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?"

- *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 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 along with the context where those expressions might invoke the method.

On a 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 relevant item. Context broadening is discussed in detail in section 6.9.

4.5 Standard solution strategies

This section describes the solution strategies that **DDP** uses to solve the above kinds of goals. These strategies follow straightforwardly once the goals and their answers have been formulated.

4.5.1 Responders queries

A responders query is written as follows:

$$[\![rcvr.sel(arg_0 \ldots arg_n)]\!]_{ctx} \star b \xrightarrow{send} ?$$

This query attempts to find the methods responding when *sel* is sent to *rcvr*, along with the context that those methods blocks respond in.

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 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.

4.5.2 Senders queries

A senders query, written $meth \xleftarrow{send} ?$, asks what message-send expressions invoke the method *meth*. The answer to this question includes a set of tuples of message-send expressions and contexts.

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*.

4.5.3 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 ⟦SmallInteger⟧; if it is a string, the inferred type is ⟦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*(Array)⟧, where *mclass*(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 $\top$ 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 method-context tuple, **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.

4.5.4 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 message-send statements.

$$\begin{array}{lcl} \langle \textit{expression} \rangle & ::= & \ldots \\ & | & \langle \textit{block} \rangle \end{array}$$

Figure 4.3: In ST_b, a block may be used as an expression.

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, then the value *exp* produces at run time will flow out of the method and into the message-send expression that invoked it. To find the one-step flow in this case, **DDP** begins by posting 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.

4.6 Blocks

In ST_0, blocks only appear as the entire body of a method. ST_b is an extension of ST_0 that allows blocks to be passed around as first-class values, just like any other objects. In ST_b, blocks may appear any place that an expression may appear, as shown in Figure 4.3.

The semantics of blocks are equivalent to the semantics of lambda expressions in functional languages. Readers unfamiliar with this construct should probably either consult a text on functional languages or skip the material on blocks in this document.

Evaluating a block yields a *closure* which holds not only the expressions of the block, but also the set of variable bindings—the *environment*—that was in effect in the block's lexical scope at the time the block was evaluated. A reference to the environment is stored in the closure in order to support expressions within the block referencing those variables. Whenever an expression within a closure refers to a variable lexically outside that block, the interpreter uses the closure's environment to locate the variable binding that was in effect when the closure was created.

Converting a block to a closure does not yet evaluate the expressions within the block. To evaluate those expressions, a further step is needed: the closure must be evaluated using the primitive `value` family of methods. There are several `value` methods available depending on the number of arguments present in the closure: `value0` for a parameterless closure, `value1` for a closure with one argument, and so on.[1] All of these methods are tagged with the primitive `value`.

When a closure is evaluated using the `value` method, the interpreter creates a new activation, sets the formal parameters in that activation to the supplied actual arguments, and begins execution with the first expression in the closure.

If the closure completes execution of all of its expressions, then the `value` primitive returns the value of the last expression in the closure as the value of the `value` method that was invoked. If, however, the closure

[1]In Smalltalk, these methods are called `value`, `value:`, `value:value`, etc.

evaluates an `return` expression, then control returns from the surrounding *method*—a non-local return—and thus the closure itself never returns a value.

When analyzing $\mathrm{ST_b}$, **DDP** includes a new kind of type, the *block type*. A block type specifies a particular block from the source code. A block type includes all closure objects which were created by evaluating the specified block. Block types additionally mention a context which matches the execution state when the block was turned into a closure. 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.

Set types, naturally, may now include block types in addition to class types.

All four kinds of queries need adjustments to account for blocks. First, responders queries now return a list of blocks that can respond in addition to the list of methods that can respond. Responding methods that invoke the `value` methods are removed from the answer to the responders query. Instead, if `closer-eval` primitive methods can respond, then the analyzer examines the type of the receiver to see which block types are included. Each block type included in the receiver type corresponds to one responder that the analyzer must include in the answer to the responders query. The block for the responder is simply the block from the block type, but the context is generated in a more sophisticated way: it is the intersection of the context from the block type with the context generated (as with responders queries in $\mathrm{ST_0}$) by inferred types for the message-send's arguments.

Senders queries may now look up the expressions that invoke a specified block; recall that senders queries for $\mathrm{ST_0}$ are always targeted at a top-level method. A different technique is used to find the senders of a block than to find the senders of a method. To find the senders of a block, **DDP** traces the flow of that block using a flow query. At each message-send expression where the object flows to the receiver, the analyzer checks whether a `value` method would respond to that message-send if a block were the receiver. If so, then the message-send is considered a potential sender of the block. The response to the responders query is the set of all such senders.

Type queries need a few small adjustments. First, the type of a block expression is a block type whose block is the block of the expression and whose context is copied directly from the query. Finding the type of a parameter, interestingly, does not need modification, because senders queries function just as well on blocks as they do on methods. Finally, the type of a message-send expression must account for the possibility that closures might invoked by the message send. For methods included in the answer to the responders query, **DDP** does the same thing as it does in $\mathrm{ST_0}$: it finds a type for each returned expression. For each closure included in the answer, **DDP** instead finds a type for the last expression in the closure. The result of a type query on a message-send expression is then the union of both the types returned by responding methods and the types returned as the last expression of evaluated closures.

Flow queries must account for flow into and out of blocks. A one-step flow query on the last expression of a block must now include in its answer all message-send expressions that might invoke the block. The expressions are found in the same way that they are found for flow from a `return` statement: a senders query is performed on the block, and the message-send expressions thus found are the answer to the flow query. When computing flow from a message-send expression, **DDP** must be careful to consider blocks that respond to the message send in addition to methods that respond. The arguments to the message send can flow into block parameters just as they flow into method parameters.

4.7 The `perform` methods

$\mathrm{ST_{bp}}$ extends $\mathrm{ST_b}$ to allow message sends where the selector is computed at run time. This facility is used in Smalltalk in GUI frameworks and in other code where a generic "pluggable" component can invoke a client-specifiable method on a target object. While blocks can accomplish the same code pattern—and in fact, **DDP**'s analysis of `perform` is similar to that for blocks—`perform` is popular due to allowing this common idiom to be significantly more concise.

In $\mathrm{ST_b}$, blocks became first-class values instead of merely syntax. Analogously, in $\mathrm{ST_{bp}}$, selectors are first-class values. Syntactically, selectors are specified as literals, as shown in **??**: a hash mark (#) followed by an identifier represents a selector.[2] Such a selector can be passed to methods tagged with a new primitive named `perform` to accomplish a message-send with a computed selector. Typical $\mathrm{ST_{bp}}$ programs have a series of methods named `perform0`, `perform1`, `perform2`, etc., in class Object, which are all tagged with primitive `perform` but have different numbers of parameters.

When one of these `perform` methods is invoked, the interpreter looks at the first argument to find the selector that is to be sent. If the first argument is not a selector object, then computation halts.[3] Otherwise, a new message is created whose selector is the first argument of the original message and whose arguments are the remaining arguments of the original message. That message is then sent to the receiver object just as if the message had been sent with a normal message-send expression.

Analyzing $\mathrm{ST_{bp}}$ requires adding a new form of types and tweaking the inference rules used for several kinds of queries. The new form of type is a *selector type*, and is simply a selector itself. For example, `#straight` is a type which includes only the selector object named `straight`.

Responders and senders queries must include in each item they answer a *parameter skip count* representing the number of invocations of `perform` that lie between the queried item and the answered item. For responders queries on some message-send expression, the parameter skip count is the number of times `perform` is invoked before the designated method responds. A skip count of 0 means that the message-send expression invoked the method directly. A skip count of 1 means that the message-send expression invoked `perform` which then invoked the designate method. Larger skip counts are rare in practice but are included for completeness. A skip count of 2, for example, means that the message send invoked `perform`, which then invoked `perform` again, and which only then invoked the designated method.

The parameter skip count for senders queries is analogous. Non-zero skip counts mean that the queried method or block is directly called by the associated message-send statement. A skip count of 1 means that the associated message-send statement invokes `perform` which in turn invokes the queried method or block. Likewise for higher counts.

The skip count is used in type and flow queries to match actual parameter with formal parameters. Instead of the ith actual parameter matching the ith formal parameter, the skip count requires that in general some of the first parameters might need to be skipped on either end. For a type query on the ith formal parameter of some method or block, a senders query is performed to find the possible senders of the method; for a sender with skip count sk, **DDP** must match the queried parameter to actual argument $i + sk$ instead of actual argument i. Likewise, for a flow query on an actual argument, **DDP** must use skip counts to match the actual arguments to the formal parameters of each possible responding method or block.

As with analyzing blocks, responders and senders queries gain the bulk of the additional complexity from adding `perform` to the language. Responders queries must check each possible responder and see if it is tagged with the `perform` primitive. If so, then that responder is not included directly. Instead, the analysis is repeated using each selector type inferred for the first argument of the message send, with the remaining arguments used as arguments in the repeated analysis. Any results found in the repeated analysis are given a parameter skip count of 1. If any of those responders are themselves `perform` primitives, then the analysis is repeated again, this time yielding responders with a skip count of 2. This iteration is repeated until no `perform` methods respond. Note that this iteration must terminate, because every message send has only a finite number of parameters, and each iteration has one less parameter than the previous.

Senders queries must, in addition to their usual analysis, account for senders via `perform`. To find these senders for a method, they create subgoals for each occurrence of the method's name as a selector literal

[2]In Smalltalk, symbols are used as selector objects, and thus selectors can be computed using arbitrary string operations. Such usage is not needed for the common "pluggable" idiom, and **DDP** ignores such usage in order to keep the analysis tractable. **DDP** treats selectors computed using string operations in the same way it treats reflective features such as the abilities to access a variable by name and to compile a class at run time.

[3]Or to be pedantic, the primitive fails, the method's fail code runs, and the fail code universally halts the computation in some form, e.g. by entering a debugger.

in the program. If there are no such occurrences, then there are no `perform` senders. Otherwise, for each message-send expression where the selector flows to the first argument, the analyzer uses a type query to check whether that message-send expression might invoke a `perform:` method. If it can, then the message-send is considered a possible sender of the method, with skip count 1.

To find senders with skip count 2, the analyzer must trace the forward flow of the appropriate `perform` selector and find message-send expressions where the selector is the first argument, intersect those with the message-send expressions where the original method's selector flows to the second argument, and then use a type query to check whether the message-send can invoke a `perform` method. Likewise for larger skip counts. Skip counts greater than 1 are rare in practice, to the extent that an analyzer writer can even be excused if they are not supported at all or if they are only supported to a small level such as 2 or 3.

Senders queries on a block must also have an answer that considers senders via `perform`. To so do, the analyzer traces the flow of block as normal, but also traces flow of the appropriate `value` selector. Message-send expressions where the `value` selector is the first argument and the block is the receiver are potential invokers of the block with a parameter skip count of 1. Senders with larger skip counts are accounted for just as with those for methods: the analyzer traces the flow of the appropriate `perform` selector (or selectors, for even higher-level skip counts) itself, and finds those message-send expressions where the `perform` selector is the first argument, the `value` selector the second argument, and the block the receiver.

4.8 Pruning

4.8.1 Overview

As discussed in Chapter 3, the P in **DDP** is that **DDP** uses pruning to keep the goal pool from growing unmanageably large. An individual goal is pruned by giving it a sufficiently conservative result that the goal no longer needs any subgoals. If the subgoal is no longer needed by the initial goal—either directly as a subgoal or indirectly by a chain of subgoal relations—then the goal is *irrelevant* to the initially posted goal. Goals that become irrelevant in this fashion can be removed from active consideration—specifically, they can be removed from `worklist` and from the `needs` network. Locating irrelevant goals requires a linear-time traversal of the `needs` network.

Pruning algorithms are heuristics, and a variety of approaches are possible. This section describes pruning algorithms usable by **DDP**. Given the above general description of pruning, it successively narrows the design space towards the specific family of pruning algorithms implemented in the prototype implementation.

4.8.2 Pruning in batches

DDP balances three kinds of work:

1. Pruning, as described above.
2. Improving approximations to goals that will, when the algorithm terminates, be used to justify of the root goal's solution.
3. Improving approximations to goals that will be pruned.

Work in the second category is the core information computation of the algorithm, while work in the last category is, in hindsight, wasted. Thus, the time allocated for pruning requires some balance. Pruning frequently means that goals are pruned sooner and work in the last category is reduced. On the other hand, pruning frequently also reduces the amount of available time for the essential work in the second category.

To balance these three kinds of work, the **DDP** prototype prunes in batches. Whenever the pruner runs at all, it scans the entire set of active goals and chooses a number of prunings. Such a pruner is requires linear time (or more) in the number of active goals. Thus, the prototype is careful to run the pruner when it has done a number of goal updates (work in the second two categories) proportional to the number of goals at the time the pruner runs.

This approach balances the work in the first two categories while limiting work in the third category. If the pruner ran any less frequently, then the algorithm could spend excessive time improving goals that will be pruned in the very next batch of pruning. If the pruner ran more frequently, then the pruner itself could dominate the algorithm's execution time.

An additional advantage of the batch-pruning approach is that the pruner can rely on computations that require linear time in the number of active goals. A simple example is that it can use a simple graph traversal to precisely identify goals that are no longer needed by the root. Other examples are given in the next section.

4.8.3 Pruning thresholds and the root-proximity heuristic

Fundamentally, the pruner must choose some goals to keep and others to prune. The prototype's pruner uses a heuristic that prefers goals which are closer to the root goal in the dependency graph. Intuitively, a direct subgoal of the root goal is more likely to influence the goal's answer than is a subgoal of a subgoal of a subgoal of a subgoal of the root goal. One would prefer to keep all subgoals, but if some must be pruned, then prune the subgoal of the sub-sub-sub-subgoal. This is the *root-proximity heuristic*. It is analogous to the heuristic in chess-playing programs that focus more attention on board positions one in the future than on board positions ten moves in the future.

The pruner in the prototype, whenever it runs, chooses to keep a number of goals that are relatively close to the root goal, while pruning all goals that are further away. It uses a *pruning threshold* to decide how many goals to keep. The choice of pruning threshold balances time expended versus quality of results. Larger thresholds mean that the algorithm tends to require more time but tends to find better answers. The experiments described in Chapter 10 shed some light on reasonable choices for the pruning threshold.

The algorithm is as follows. First, choose the goals to keep: keep the *pruning-threshold* goals that are closest to the root goal. Second, prune all kept goals that depend on any goal not chosen to be kept. Finally, discard from active consideration all goals which are not needed, directly or indirectly, by the root goal.

As a small modification, the prototype uses a modified definition of "distance from the root goal." The pruner still chooses the *pruning-threshold* goals closest to the root goal, but the definition of *closest* is modified. The modified definition gives some subgoal relationships a larger distance than others. Most subgoals are given a distance of 1, but senders-of and responders-to subgoals are given a larger distance of 4. The intuition behind this idea is that pruning other kinds of goals tends to give a poor solution not only to the pruned goal, but to an entire chain of goals closer to the root goal. Pruning a call-graph subgoal is less prone to ruining the chances for the goal depending on it. This modification appears to help based on informal trials, but has not been thoroughly tested empirically.

The choice of pruning threshold is an empirical matter discussed in Chapter 10. To summarize the empirical work, a pruning threshold of 50 gives a fast but moderately precise analyzer, while a threshold in the range of 2000-3000 yields a precise but only moderately fast algorithm. Larger thresholds than 3000 continue to increase the time required for analysis without giving much improvement in precision.

Note that a pruning threshold is different from a *pruning depth*. A pruning depth specifies a maximum distance from the root goal directly. The pruner would select all goals within the distance of the root goal, regardless of how many goals are selected.

While there is no empirical evidence comparing pruning depths to pruning thresholds, the prototype uses pruning depths because, intuitively, they seem likely to give more reliable control over the time expended and the quality of the results. The main source of intuition, here, is that a fixed pruning depth can result in a large variation in the number of goals explored, depending on the initial query, which surely leads to higher variation in the time required. A fixed pruning threshold, to contrast, results in a fixed number of goals being explored at a time by the algorithm.

4.8.4 Shrinking the threshold for real-time response

Looking ahead, the empirical study in Chapter 10 shows that the pruning threshold gives effective control of the time of execution, but only on average. A fixed threshold yields a high variance in the required time.

The pruning threshold need not be fixed. It can adaptively decrease over time if the algorithm appears to be requiring too much time. As a result, real-time response is possible. Such an approach is fleshed out in **??**, using choices based on the empirical results.

4.8.5 Drop-dead pruning

The *Stop Dead* pruning algorithm is an alternative algorithm that is simpler but empirically less effective. Instead of having a pruning threshold, Stop Dead has a *time limit*. Stop Dead performs no pruning at all until the time limit is reached. At that time, the root goal itself is pruned.

4.9 Other language features

This chapter thus far has described **DDP** as an analysis of a subset of Smalltalk. This chapter describes modifications sufficient to address a number of features present in Smalltalk.

4.9.1 Primitive methods

Smalltalk has *primitive methods* in addition to normal methods. A primitive method has the usual attributes of a method, plus additionally a reference to some *primitive routine* in the underlying interpreter. Whenever a `send` or `sendvar` method invokes a primitive method, the designated primitive routine is executed instead of the block of the method. If the routine is successful, then control passes directly back to the `send` or `sendvar` statement. If the routine is not successful, then the block of the method executes after all, just as if the method were not primitive. The block is called the *fail code* of the method, because it is only executed if the interpreter routine has not succeeded for some reason.

Correct analysis requires that the analysis account for the possible execution of primitive methods. For each primitive, there are four possible approaches:

- Use the general framework described below.
- Use a conservative approximation suitable for any well-behaved primitive routine.
- Include specialized support for the method.
- Ignore the primitive routine.

The first approach is to provide the following information to the algorithm:

- `typeWhenSentTo:withArgumentTypes:`, a function mapping a list of argument types and a receiver type to the type returned by the routine.
- `canFailWhenSentTo:withArgumentTypes:`, a function mapping a list of argument types and a receiver type to a boolean designating whether the primitive can fall into the method's fail code when invoked with objects of the specified types.
- `receiverEscapes`, a function designating whether the receiver can flow to an arbitrary flow position after this routine is invoked.
- `argumentEscapes:`, a function designating whether a particular argument can flow to an arbitrary flow position after the routine is invoked.

Adjusting the solution strategies to account for primitive methods is straightforward when they can be accurately described with the above four attributes. The strategies for type queries on message-send expressions need to check the contribution of the primitive to the type returned by a method, and they need to ignore return statements in the method if the primitive does not fail with the supplied arguments. The queries on flow via message sending need to check whether flow into any invoked primitive method can escape; if so, then the target of the relevant flow judgements must be $\top_{fp}$.

The second approach is to use a single conservative approximation for all of the above properties: the return type is $\top$, the routine can fail regardless of the argument types, and the receiver and arguments have escaping flow. So long as the primitives are well behaved, this approximation will yield correct results. As a few examples, well behaved primitives may not modify the stack of running activations, invoke some other method, or modify the program; on the other hand, well behaved primitives may access external state and create new objects that do not have instance variables.

The third approach is to add specialized support to the implementation. The rare cases that such support are necessary are described in **??**.

Finally, some primitive routines may be safely ignored. Certainly, if the primitive is known to be only an optimization of the method's fail code, then the primitive may be ignored. Additionally, a method's primitive may be ignored if, for some reason, the method is assumed never to be invoked. In this case the correctness of the analysis depends on whether the method is invoked. It is better to have fewer methods assumed not to be invoked, but it may be impossible to reduce the number to zero. Some methods, such as those for constructing a program and those for debugging a process, are simply too difficult to model effectively.

4.9.2 Instance creation

The `new` statement in $\mathrm{ST}_{\mathrm{bp}}$ is implemented in full Smalltalk as a primitive method called `#basicNew`. This primitive is well behaved, in the meaning specified in the previous section, and thus it can be supported by **DDP** with the usual mechanism for handling primitives.

To see that this approach results in precise analysis, one must consider the way classes are represented in Smalltalk. In Smalltalk, classes are normal objects. All objects have a class, and the class of a normal class object is another class object called a *metaclass*. (The class of a metaclass is the normal class MetaClass.) Each metaclass has a single instance throughout the execution of any Smalltalk program. Thus the `typeWhenSentTo:withArgumentTypes:` method for the `#basicNew` primitive can determine with accuracy which class is being instantiated: the type of a class will be a class type where the class is a metaclass, and each metaclass has only a single instance, in this case the class that is about to be instantiated.

4.9.3 Multiple processes

$\mathrm{ST}_{\mathrm{bp}}$ programs have a single process. Full Smalltalk programs may start new processes by invoking the `#fork` primitive method on a block object. The analysis needs to account for the fact that the statements in the block may execute even though there is no direct flow of control to the statements other than through the `#fork` method.

In fact, the justification rules of Chapter 7 do account for the execution of such statements. Senders-of goals to methods invoked by statements in the block, will consider message-send statements in the block as possible senders. Type goals for variables assigned in the block, will account for those assignment statements. Flow goals for variables accessed in the block will account for flow due to those statements executing.

Note that the situation is different for any Smalltalk implementation where processes are started with blocks that have one or more arguments. Then, statements that access the parameter of the block, may try to find a type judgement on one of the parameters of the block. Without care, those type judgements might judge the type to be $\perp$. In such a Smalltalk, the justification rule for the type of a parameter of a block, must also check whether the block ever flows into `#fork` primitive method; if so, then an appropriate type—possibly $\top$—must be unioned to the type judged for the parameters of the block.

4.9.4 Exceptions

Standard Smalltalk implementations include exceptions. Smalltalk exceptions are more flexible than in many languages with exceptions, in that an exception handler can choose to resume the context that signaled an exception. If they choose to resume it, they can even pass an object to the exception-raising context. Thus, data can flow in both directions when an exception is raised and handled. As an example of the power of this feature, note that it is sufficient for implementing dynamically bound variables: dynamically bound variables are exception types, bindings of such a variable are exception handlers, and accesses to those variables are raises of the exception type.

Control- and data-flow through exceptions can be analyzed precisely [61], but the analysis is complicated. Since typical programs only use exceptions for the purpose of short-circuiting control-flow paths—and rarely, for example, to implement dynamic variables—it should suffice to implement a conservative approximation.

A working conservative approximation for data flow is to assume that any value passed into an exception's methods can flow to anywhere in the program, and that any value read via an exception's methods can be of any type. Given this conservative data-flow approximation, nothing needs to be added to control flow. The control-flow queries of **DDP** are queries about the call graph. Control paths involving exceptions are never asked about and thus do not need to be computed.

Naturally, this conservative approximation can be refined if one does add control-flow queries for exception handling. One could add the queries "which handler responds to this exception-raise" and "which exception-raises might have invoked this handler." Given such queries, one could then add more precise data-flow computations. This avenue has not been explored by the authors.

4.9.5 Message sends to `super`

In addition to regular message sends, Smalltalk, like most object-oriented languages, includes "`super`" message sends. A `super` send uses a modified message-lookup algorithm from the standard **lookup** function described in section 5.11. Instead of searching for a responding method by starting at the responding object's class, it begins by searching at the superclass of the method where the `super` send appears. This lookup can be performed statically with perfect precision.

Modifying **DDP** to account for `super` sends is straightforward. Responders queries on a `super` send simply use the static lookup algorithm. They do not need to perform a type query on the receiver. Senders queries on a method must consider `super` sends in addition to regular message sends. To do so, they must add to their response all `super` sends which might, according to the static lookup algorithm, invoke the method in question. Senders queries on a block need no modification, because `super` sends only invoke methods.

4.9.6 Initial state

$\mathrm{ST}_{\mathrm{bp}}$ programs begin with an empty heap and start execution at a specific method. Full Smalltalk programs begin with a populated heap of objects and with a number of running processes. The justification rules need to be adjusted so that they account for this initial state. The required modification is simple: adjust the solution strategy for type queries to include, in addition to the type found by the usual solution strategies, the types found by scanning the initial heap and stacks.

4.9.7 Arrays and other collections

Arrays in Smalltalk are supported by a combination of two implementation features: class definitions may include a declaration that a class is *indexable*, and there are primitive methods `#at :` and `#at : put :` available to read and write to the indexed slots in an instance of such a class.

DDP handles arrays conservatively by ignoring the *indexable* declaration, and by treating the `#at :` and `#at : put :` methods pessimistically: they might return a value of any type, and any object that is `#put :` into an indexed slot flows to $\top_{fp}$.

4.9.8 Array literals and `sendvar`

Literal statements in full Smalltalk may create arrays that contain selector objects. Since the contained selector objects might eventually be used by a `sendvar` statement, it is important for the analysis to account for them. A conservative approach is to maintain an extra table named `arrayLiteralsWithSelector` in addition to the other tables described in **??**. This table gives all array literal statements in the program that contain a specified selector. The analysis uses this table to determine whether it can accurately predict what statements will invoke a particular method; if the method's selector appears in *any* array literal in the program, then it is impossible to accurately find the senders and thus the method should be given a sender set of $\top_s$.

4.9.9 Flow of literals

Many literals in Smalltalk break the assumption of $\mathrm{ST_{bp}}$ that each evaluation of a literal expressions yields a different object. Every evaluation of `true`, for example, yields the exact same `true` object in full Smalltalk. As a result, justified flow judgements in **DDP** do not account for the full flow possibilities of these values. It is possible for these values to *flow*, in the carefully defined sense of Chapter 6, in other ways than through assignment statements, message sending, and so on. It is possible for them to appear at remote locations in the program that are unrelated by the usual flow calculations.

Such a situation is beyond the supported usage of **DDP**, however. Under normal usage, flow is only computed for blocks and selectors, not for any other values, much less values that appear as literals. Each block appears only once in a program, and thus it is immune to the possible problem under discussion. For symbols, the standard **DDP** solution strategies simultaneously find flow for all occurrences of a symbol of interest, thus eliminating the possibility that a symbol literal will appear in a location outside of the computed flows.

The current theory of flow and of literal expressions has been chosen for intuitiveness and brief description. Under normal usage of **DDP**, even on full Smalltalk, it provides an accurate description. It would be possible, instead, to allow $\mathrm{ST_{bp}}$ literal expressions to evaluate to the same object multiple times, while redefining the correctness criterion of flow judgements to ignore such appearances, and thus obtain a more complicated theory that is closer to full Smalltalk. However, the added complication has been deemed too high a price for the payoff in extra assurance from the theory.

4.10 Implementation issues

This section describes some issues that arise during practical implementation of **DDP**.

4.10.1 Maintaining tables about syntax

The solution strategies given above frequently require scanning the program to find statements matching some criterion. For example, finding the type of a variable requires finding all statements that assign a value to the variable. To make such scanning fast, implementations should pre-compute and maintain the following tables:

- `methodsImplementing`, which maps each possible selector to the list of methods in the system that implement the method. This is useful in responders-to goals where the receiver type is $\top$.
- `expressionsSending`, which maps each selector to the list of `send` statements that send the selector. This is useful for finding the `send` statements that may invoke a method.
- `selectorLiterals`, which maps each selector to the list of statements assigning that selector to a variable. This is useful for finding the `sendvar` statements that may invoke a method.
- `assignmentsDefining`, the list of statements that assign something to a variable. This is useful for type goals.

- `expressionsReading`, the list of statements that read from a variable. This is useful for flow goals.

These tables must be updated whenever the programmer modifies the code base. The code changes are straightforward and, in most cases, are proportional to the amount of code that is affected by the change. For example, if the programmer adds a new method is added to the code base, then:

- The method's parse tree is added to `parseTrees`.
- The method is added to the list in `methodsImplementing` corresponding to the method's name.
- For each message-send expression in the method, the expression is added to the appropriate list in `expressionsSending`.
- For each literal expression in the method where the literal is a symbol, the expression is added to the symbol's list in `symbolLiterals`.
- For each assignment statement in the method, the statement is added to the list of statements in `assignmentsDefining` corresponding to the variable on the left-hand side of the assignment statement.
- For each variable expression, the expression is added to the appropriate list in `expressionsReading`.

These changes are reversed when a method is removed. All other code changes are described as a combination of method removals followed by method additions. Examples of such changes are:

- If a method is changed, then the tables are updated as if the method was removed followed by the method being added.
- If a class definition is changed, then the tables are updated as if every method in the class and its subclasses were removed before the definition change and added back after the definition change.

The time taken for these updates are proportional to the existing overhead of a Smalltalk dynamic byte-code compiler, which already is invoked after every code change. For example, when a new method is added, Chuck updates its tables as described above, and the system compiles that method to byte code. The additional overhead involved in maintaining the syntactic data structures is imperceptible.

4.10.2 Parse tree compression

Storing expanded parse trees for every method in the system takes considerable storage, both in number of bytes and number of objects. Yet, each execution of **DDP** is likely to access only a small fraction of all the parse trees in the image. In-memory parse tree compression is a useful technique to address these two observations: it lowers the number of bytes required, greatly lowers the number of objects required (because the compressed parse trees can be stored as binary arrays), and, because few parse trees are needed per execution, typically does not cause a large slowdown.

The general approach is for the lookup tables to refer to expressions and methods indirectly. Instead of holding object references to the expressions and methods, they hold a reference to a tuple of the method's specification (class plus selector) plus the integer index of the expression within the method. Decompressed parse trees can be cached during each execution of **DDP** so that they are only decompressed once per execution.

4.10.3 Supporting external source code

Ideally, the implementation can analyze code regardless of whether it is the code of the currently running image. If an implementation is flexible, then it supports interactive programming tools both for the installed code and for code stored outside the image. Additionally the test cases for the implementation are able to test the analysis against entire code bases that have been carefully crafted to exercise the implementation.

One challenge of such a flexible implementation is that there are multiple classes named Object, multiple classes named Block, and so on—one class for each code base that is potentially analyzed. In order to access the particular class relevant to the current execution of **DDP**, many methods must have a parameter that specifies the list of classes for this execution even though they do not have such a parameter in the on-paper description of **DDP**. The number of such methods can be reduced, however, if class BlockType holds a reference to the appropriate Block class for the current execution, and the SelectorType class likewise holds a reference to the appropriate Selector class.

Chapter 5

Mini-Smalltalk

This chapter begins the formal description of the **DDP** algorithm. This chapter defines the syntax and semantics of *Mini-Smalltalk*, the language analyzed by the formal version of **DDP**. Adaptation of **DDP** for the full Smalltalk language is described in **??**.

5.1 Overview

The present work defines a semantics tuned for giving an accurate description and a proof of correctness of the more interesting parts of the type-inference algorithm. Since Smalltalk has such a simple semantics, it seems worthwhile to spend a few pages describing a semantics tuned for the present purposes in exchange for simplifying the rest of the work. This semantics includes:

- The essential parts of a class-based, object-oriented language, including classes, objects, messages, and methods.
- Single inheritance of classes.
- Blocks with full closure semantics.
- Nested mutable variables within blocks. This feature greatly increases the complexity of the semantics and has thus been omitted from other authors' analogs to Mini-Smalltalk. Since the feature can introduce subtle errors into a program analysis, it is included in Mini-Smalltalk despite the complexity it entails.
- The `perform:` primitive, called `sendvar` in Mini-Smalltalk, which allows invoking a method with a computed name. This feature is non-trivial to support and also allows for subtle analysis errors.

Several features are omitted because they increase the complexity of the semantics but do not provide new insight.

- Arrays.
- Primitive methods such as addition and input/output.
- Processes.
- Classes as full-fledged objects.

The implementation supports these features in straightforward manners described in **??**.

Some reflective features are omitted because they are primarily intended to be used in the development environment and because supporting them is beyond the scope of most program analyses. Examples include the object-inspection tool that can modify objects in arbitrary ways, the ability to reference instance variables by name (`#instVarAt` : and `#instVarAt` : `put` :), and the `thisContext` facility for accessing the call stack.

5.2 Terminology

When discussing syntax, this paper uses Smalltalk terminology, thus keeping Mini-Smalltalk syntax close to that of Smalltalk. For example, *block statement* is used instead of *lambda expression*. On the other hand, when discussing semantics, the paper uses common terminology of the semantics literature. For example, *closure* is used instead of *block*.

5.3 Language overview

Mini-Smalltalk is a language that captures the essence of Smalltalk [8]. It includes the Smalltalk features used in application-level programming, but it does not include introspective features intended for use by the compiler or debugger. It also includes some differences from Smalltalk that simplify the theory without removing any power:

1. There are no compound expressions. Instead, there are sequences of simple statements that use temporary variables to store intermediate results.

2. Distinctions among class, pool, and global variables are ignored. Instead, they are all treated as global variables. The distinctions are unimportant for analysis because they only affect visibility and otherwise have the same semantics.

3. Classes are not values. Instead, `new` is a syntactic form.

4. There are no return statements (designated with ˆ or $\uparrow$ in Smalltalk). Instead, every block, including the main block of a method, must include a variable to return and an indication of whether the value should be returned from the current block or from the surrounding method.

5.4 Syntax

The abstract syntax of Mini-Smalltalk is given in Figure 5.1.

A Mini-Smalltalk program consists of a set of global variables and a finite map from class names to *classes*. Each class has an optional superclass, a set of instance variables, and a finite map from method names to methods. Each method has a block, called the *main block* of the method. Each block has a number of parameters, a number of local variables, and a number of statements. When the block finishes executing, it returns a value either to the statement that invoked the block or (non-locally) to the statement that invoked the surrounding method.

Each statement has one of the following forms:

- l := `self`. This statement assigns the current receiver to variable l.

- l := *literal*. This statement assigns a literal, such as `4` or `'hello world'`, to a variable.

- l := l_r. This statement assigns the contents of one variable to another variable.

- l := `new` *classname*. This statement instantiates a new object of the class named *classname*.

- l := *block*. This statement creates a closure, just like a lambda expression in Scheme.

- l := `send`(*rcvr*, *selector*, $arg_1 \ldots arg_m$). This statement sends a message to *rcvr*. The expression requests that a method matching *selector* will execute, and it supplies $arg_1 \ldots arg_m$ as parameters to the method.

$$
\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \text{Program} \\
 & & \quad \text{globals: } \langle \textit{label} \rangle * \\
 & & \quad \text{classes: } (\langle \textit{label} \rangle \, \langle \textit{class} \rangle) * \\
 & & \\
\langle \textit{class} \rangle & ::= & \text{Class} \\
 & & \quad \text{superclass: } (\langle \textit{label} \rangle \, | \texttt{undef}) \\
 & & \quad \text{methods: } (\langle \textit{label} \rangle \times \langle \textit{method} \rangle) * \\
 & & \quad \text{instance variables: } \langle \textit{label} \rangle * \\
 & & \\
\langle \textit{method} \rangle & ::= & \text{Method} \, \langle \textit{block} \rangle \\
 & & \\
\langle \textit{block} \rangle & ::= & \text{Block} \\
 & & \quad \text{parameters: } \langle \textit{label} \rangle * \\
 & & \quad \text{temporaries: } \langle \textit{label} \rangle * \\
 & & \quad \text{statements: } \langle \textit{statement} \rangle * \\
 & & \quad \text{returning: } \langle \textit{label} \rangle \\
 & & \quad \text{retFromMethod: } \langle \textit{boolean} \rangle \\
 & & \\
\langle \textit{statement} \rangle & ::= & \langle \textit{label} \rangle := \texttt{self} \\
 & | & \langle \textit{label} \rangle := \langle \textit{literal} \rangle \\
 & | & \langle \textit{label} \rangle := \langle \textit{label} \rangle \\
 & | & \langle \textit{label} \rangle := \texttt{new} \ \langle \textit{label} \rangle \\
 & | & \langle \textit{label} \rangle := \langle \textit{block} \rangle \\
 & | & \langle \textit{label} \rangle := \texttt{send}(\langle \textit{label} \rangle, \langle \textit{selector} \rangle, \langle \textit{label} \rangle *) \\
 & | & \langle \textit{label} \rangle := \texttt{sendvar}(\langle \textit{label} \rangle, \langle \textit{label} \rangle, \langle \textit{label} \rangle *) \\
 & | & \langle \textit{label} \rangle := \texttt{beval}(\langle \textit{selector} \rangle, \langle \textit{label} \rangle *) \\
 & & \\
\langle \textit{selector} \rangle & ::= & \text{Selector} \\
 & & \quad \text{label: } \langle \textit{label} \rangle \\
 & & \quad \text{numargs: } \langle \textit{integer} \rangle
\end{array}
$$

Figure 5.1: Abstract Syntax of Mini-Smalltalk

- l := `sendvar`(*rcvr*, *selectorvar*, $arg_1 \ldots arg_m$). This statement also sends a message, but the selector of the method to invoke is read from *selectorvar*. This statement supports the `#perform` : functionality of Smalltalk, although notice that in Mini-Smalltalk, the only way to create a selector object is via a literal statement. There is no method in Mini-Smalltalk to convert a string to a selector. The analysis assumes that the program does not use any such feature that is present, just as it assumes the program uses no introspective debugging features.

- l := `beval`(*blockvar*, $arg_1 \ldots arg_m$). This statement reads a closure from the variable named *blockvar* and invokes it.

Notice that some elements of syntax in Mini-Smalltalk are primitive methods in full Smalltalk. To analyze such constructs, implementations of **DDP** should treat such primitive methods as if they contained a single statement with the corresponding syntactic element from Mini-Smalltalk. For example, any method that references the block-evaluation primitive would be treated as the following Mini-Smalltalk method:

```
value

    | block result |

    block := self.

    result := beval block.

    return result.
```

Similarly handled are the primitive methods for `sendvar`. The primitive method for `new` is handled differently, as described in the previous section.

As a matter of notation, an expression like *foo.bar* refers to the *bar* component of *foo*. For example, if

$$\mathcal{P} = \text{Program globals: } g \text{ classes: } c$$

then $\mathcal{P}.globals = g$ and $\mathcal{P}.classes = c$.

5.5 Concrete syntax for methods

The abstract syntax is convenient for mathematics but cumbersome for manipulation of large amounts of code. A concrete syntax for methods is summarized in Figure 5.2. The concrete syntax is more convenient for the larger amounts of code given in examples and is closer to the syntax of full Smalltalk.

5.6 Valid programs

A program $\mathcal{P}$ is a *valid program* if it has the following properties:

1. All variable labels are different from each other. This causes no loss of generality because Mini-Smalltalk is lexically scoped. If two variables have the same label, then one or the other may be renamed without changing the meaning of the program.

2. $\mathcal{P}.classes$ includes a class UndefinedObject. That class has no instance variables, and it does have a method with selector `#DoIt`.

3. $\mathcal{P}.classes$ includes two more classes Block and Selector which have no instance variables.

4. The class hierarchy is acyclic: no non-empty chain of *superclass* attributes will link a class back to itself.

$$
\begin{array}{rcl}
\langle \mathit{method} \rangle & ::= & \langle \mathit{header} \rangle \, \langle \mathit{block\_body} \rangle \\
\langle \mathit{header} \rangle & ::= & \langle \mathit{unary\_selector} \rangle \\
 & | & (\langle \mathit{keyword} \rangle \, \langle \mathit{identifier} \rangle) * \\
\langle \mathit{block\_body} \rangle & ::= & (\text{``}|\text{''} \, \langle \mathit{identifier} \rangle * \text{``}|\text{''}) \\
 & & (\langle \mathit{statement} \rangle \, \text{``.''}) * \\
 & & \text{``\^{}''}? \, \langle \mathit{identifier} \rangle \\
\langle \mathit{statement} \rangle & ::= & \langle \mathit{identifier} \rangle \leftarrow \texttt{self} \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \langle \mathit{identifier} \rangle \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \langle \mathit{literal} \rangle \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \texttt{new} \; \langle \mathit{identifier} \rangle \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \text{``[''}(\text{`` :''} \, \langle \mathit{identifier} \rangle) * \langle \mathit{block\_body} \rangle \, \text{``]''} \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \langle \mathit{identifier} \rangle \, \langle \mathit{unary\_selector} \rangle \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \langle \mathit{identifier} \rangle \, (\langle \mathit{keyword} \rangle \, \langle \mathit{identifier} \rangle) + \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \langle \mathit{identifier} \rangle \, \texttt{perform:} \, \langle \mathit{identifier} \rangle \, (\texttt{with:} \, \langle \mathit{identifier} \rangle) * \\
 & | & \langle \mathit{identifier} \rangle \leftarrow \texttt{beval} \, \langle \mathit{identifier} \rangle \, (\texttt{with:} \, \langle \mathit{identifier} \rangle) *
\end{array}
$$

Figure 5.2: Concrete syntax for methods of Mini-Smalltalk

5. For all literals *lit* in literal statements, the class of the literal is included in $\mathcal{P}.classes$. Formally, **lit_classes**(*lit*) $\subseteq \mathcal{P}.classes$

6. Every `send` statement supplies the exact number of arguments that the statement's specified selector requires.

7. Every method has the same number of parameters as the method's selector requires.

Programs in this paper are implicitly assumed to be valid.

5.7 Literals

The precise forms that a literal may take are left unspecified, because those details have no impact on the overall srtucture of the type inferencer described in this document. A function **inst_literal**, described in section 5.11, is used to instantiate new literals as a program executes. Further constraints on literals are described in section 6.3.

5.8 Method specifications and block specifications

The semantics include two new structures that refer to elements of the program being executed: method specifications and block specifications.

A *method specification* refers to a method from the source program. Its attributes are:

- *class_name*, the name of the class to which the method belongs
- *selector*, the selector of the method

A *block specification* refers to a block from the source program. It specifies a method plus a navigation path through the statements of the method to find a block at an arbitrary level of nesting. Its attributes are:

- *method*, the method specification for the method containing the block.
- *statement_nums*, a sequence of integers corresponding to statement numbers. An empty sequence designates the main block of the method. A one-element sequence $[i_1]$ designates the block created by the i_1-th statement of the main block (which must be a block statement). A two-element sequence $[i_1, i_2]$ designates the block created by statement i_2 of the block created in statement i_1 of the main block. Likewise for longer sequences.

Some blocks are nested within others, which gives rise to an ordering among block specifications: $b_1 \sqsubseteq b_2$ when b_1 is nested within b_2, as described in Figure 5.3. Additionally, block specifications may be combined in a simple fashion, as described in Figure 5.4 and Figure 5.5. Note that $\top_{bs}$ and $\bot_{bs}$ elements have been added in order to complete a lattice; such specifications are meaningless and are included only to simplify the mathematics.

5.9 Functions over syntax

The set **all_blocks**($\mathcal{P}$) includes all block specifications in $\mathcal{P}$. It includes the blocks of the methods of $\mathcal{P}$, and it recursively includes any blocks in block statements within the set. The block associated with block specification *bs* in $\mathcal{P}$ is designated $\mathbf{block}_{\mathcal{P}}(bs)$.

5.10 Semantic structures

This section defines semantic data structures used during the execution of a program.

A *contour* binds a set of variables. All variable bindings are held in contours in order to support mutation of variables by the various assignment statements that Mini-Smalltalk includes. A contour is a finite map from labels to objects. A contour is referred indirectly via a *contour id* or *cid*. There are two distinguished contours: `NilCID`, the contour of the distinguished object `nil`, and `GlobalsCID`, the contour used to bind global variables.

An *object* has a *class* that names the object's class, and an *ivars_cid* identifying the contour that holds the object's instance variables.

There are three kinds of objects:

- A *normal object*, which is created by either a `new` statement or by a literal. The class of such an object may be any class other than Block or Selector.
- A *closure* is created by a block statement. A closure's class is always Block. It has two attributes other than the usual ones for objects:
  - *sblock* is the block statement in the statement that created the closure.
  - *outer* is the activation (defined below) in which the *sblock* block statement was executed. This information is needed in the semantics of non-local returns and lexically scoped variable access.
- A *selector object* is created by a literal statement where the literal specifies a selector. Selector objects have class Selector and are distinguished by their *label* and *numArgs*. They may have no instance variables.

Selector objects and closures must always have a contour id that references an empty contour.

The distinguished object `NilObj` is an instance of class UndefinedObject. Its contour id is `NilCID`, which will always reference an empty contour.

An *activation* is the current state of execution for one closure or method. It is analogous to a stack frame in a typical language implementation. An activation has the following attributes:

BSO-NESTED
$$\frac{}{(ms, l@l') \sqsubseteq (ms, l)}$$

BSO-TOP
$$\frac{}{bs \sqsubseteq \top_{bs}}$$

BSO-BOTTOM
$$\frac{}{\bot_{bs} \sqsubseteq bs}$$

Figure 5.3: Comparison of Block Specifications

BSJ-SYM
$$\frac{bs_1 \sqcup bs_2 = bs_3}{bs_2 \sqcup bs_1 = bs_3}$$

BSJ-TOP
$$\frac{}{bs \sqcup \top_{bs} = \top_{bs}}$$

BSJ-BOTTOM
$$\frac{}{bs \sqcup \bot_{bs} = bs}$$

BSJ-DIFFMETH
$$\frac{ms_1 \neq ms_2}{(ms_1, l_1) \sqcup (ms_2, l_2) = \top_{bs}}$$

BSJ-SAMEMETH
$$\frac{l = \textbf{longest\_prefix}(l_1, l_2)}{(ms, l_1) \sqcup (ms, l_2) = (ms, l)}$$

Figure 5.4: Join for Block Specifications

BSM-SYM
$$\frac{bs_2 \sqcap bs_1 = bs_3}{bs_1 \sqcap bs_2 = bs_3}$$

BSM-TOP
$$\frac{}{bs \sqcap \top_{bs} = bs}$$

BSM-BOTTOM
$$\frac{}{bs \sqcap \bot_{bs} = \bot_{bs}}$$

BSM-DIFFMETH
$$\frac{ms_1 \neq ms_2}{(ms_1, l_1) \sqcap (ms_2, l_2) = \bot_{bs}}$$

BSM-DIFFBLOCK
$$\frac{a \neq b}{(ms, l@[a]@l_1) \sqcap (ms, l@[b]@l_2) = \bot_{bs}}$$

BSM-NESTED
$$\frac{}{(ms, l) \sqcap (ms, l@l') = (ms, l@l')}$$

Figure 5.5: Meet for Block Specifications

- *block_spec*, a specification for the block that is executing
- *pc*, the index of the next statement in the block to execute
- *caller*, the activation that sent the message that created this activation, or `undef`
- *outer*, the activation where temporary variables from one lexical scope outward should be looked up, or `undef` if there is no such activation
- *receiver*, the receiver object to which the message was sent
- *params_cid*, a label for the contour holding this block's parameters
- *temps_cid*, a label for the contour holding this block's temporary variables
- *caller_var*, the variable into which the return value should be placed

A *configuration* is a tuple $(activation, contours)$. The *activation* is either the currently active activation or the value HALTED. The special value HALTED means that execution is *halted*, either because execution has completed or because there has been some dynamic error such as sending a message to an object that does not understand it. In this semantics, execution never becomes stuck—instead, execution enters the HALTED state and never leaves it.

The *contours* part of a configuration tuple is a mapping of contour ids to contours. It holds the current values referred to by all variables.

Not all objects are sensible to discuss for a particular program and configuration. A *valid object* for a program $\mathcal{P}$ and configuration *cfg* must follow some additional restrictions. First, its *class* must name one of the classes in $\mathcal{P}$. Second, its *ivars_cid* must be among the contours of *cfg*. Third, the domain of the specified contour must be precisely the instance variables of the object's class in $\mathcal{P}$, including instance variables that have been inherited. Finally, if the object is a closure, then the activation of the closure's block must be a valid activation for $\mathcal{P}$ and *cfg*.

5.11 Semantic functions

This section defines the low-level functions upon which the semantics is built.

The set **all_objects**(*cfg*) includes all objects in use in configuration *cfg*, and **all_activations**(*cfg*) is the set of all activations that are accessible in configuration *cfg*. The two functions are mutually recursive. The base cases are that **all_objects**(*cfg*) includes all objects in the range of any of *cfg*'s contours, and **all_activations**(*cfg*) includes the current activation of *cfg*. The inductive cases are that **all_objects**(*cfg*) includes the receiver of any one of the activations in **all_activations**(*cfg*), **all_activations**(*cfg*) includes the *outer* and *caller* of any activation in **all_activations**(*cfg*), and finally **all_activations**(*cfg*) includes the activation of any block object in **all_objects**(*cfg*).

$\mathbf{lookup}_{\mathcal{P}}(C, sel)$ looks up a method in a specified class, given the selector for that method. It returns either a single method or `undef`. It is defined recursively as follows:

- If $sel \in \mathbf{domain}(\mathcal{P}.classes[C].methods)$, then $\mathcal{P}.classes[C].methods[sel]$.
- Otherwise, if $\mathcal{P}.classes[C].superclass =$ `undef`, then `undef`.
- Otherwise, $\mathbf{lookup}_{\mathcal{P}}(\mathcal{P}.classes[C].superclass, sel)$.

The function **inst_literal** instantiates a literal. Its arguments are a syntactic literal and a configuration *cfg*. It returns an object and a new configuration. The new configuration is identical except that a new contour has been added; the object's contour id refers to the new contour. The new object *must* be a new one; it must use a contour id that is previously unused.

lookup_contour$((act, cnt), act_v, label, allowparam)$ =

```
if       $label \in \textbf{domain}(cnt[(act_v.temps\_cid)])$
         then $act_v.temps\_cid$
else if  $label \in \textbf{domain}(cnt[(act_v.params\_cid)])$
         then (if $allowparam$ then $act_v.params\_cid$ else undef)
else if  $act_v.outer \neq$ undef
         then $\textbf{lookup\_contour}_{\mathcal{P}}((act, cnt), act_v.outer, label, allowparam)$
else if  $label \in \textbf{all\_instvars}_{\mathcal{P}}(act_v.rcvr.class)$
         then $act_v.rcvr.ivars\_cid$
else if  $label \in \mathcal{P}.globals$
         then GlobalsCID
else     undef
```

Figure 5.6: Looking up the contour that binds a variable label. (act, cnt) is the configuration in which to look up the variable, act_v is the activation in which to look up the variable, *label* is the variable's name, and *allowparam* specifies whether the function should succeed if the variable binds to a parameter. *allowparam* is used to support parameters being read-only.

$$\frac{\begin{array}{c} cid = \textbf{lookup\_contour}_{\mathcal{P}}(cfg, act_v, label, \text{true}) \\ contour = cfg.contours[cid] \end{array}}{contour[label] = \textbf{read\_var}(cfg, act_v, label)}
\qquad
\frac{\begin{array}{c} cid = \textbf{lookup\_contour}_{\mathcal{P}}(cfg, act_v, label, \text{false}) \\ contour = cfg.contours[cid] \\ contours' = contours[cid \mapsto contour[label \mapsto object]] \\ cfg' = (act_v, contours') \end{array}}{cfg' = \textbf{write\_var}(cfg, label, object)}$$

Figure 5.7: Reading and writing variables

The function

$$\textbf{lookup\_contour}_{\mathcal{P}}(cfg, act, label, allowparam)$$

searches for the contour that binds a specified variable. The last parameter specifies whether contours for parameters should be returned. The function is defined in Figure 5.6.

The function **read_var**$(cfg, act_v, label)$ returns the object that a specified variable holds in a specified configuration. Note that the activation to read from is specified via the act_v parameter. While the semantics itself will always use the main activation of *cfg*, the generalized definition of **read_var** will later prove useful for stating stronger invariants about variable contents. The function **write_var**$(cfg, label, object)$ writes a new object into a variable and returns the resulting configuration. Both **read_var** and **write_var** are defined in Figure 5.7.

The initial configuration for $\mathcal{P}$ is denoted $\textbf{step}_0(\mathcal{P})$. Likewise, $\textbf{step}_n(\mathcal{P})$ represents the program after n applications of **step** to the initial configuration.

5.12 Initial configuration

The semantics of Mini-Smalltalk will be described operationally. This section describes the initial configuration for any particular program, and the next section describes the **step** function which moves one configuration to the next. The initial configuration is a tuple $(\mathit{activation}_0, \mathit{contours}_0)$ defined as follows.

Let *startmeth* represent the start method of the program:

$$\mathit{startmeth} = \mathbf{lookup}_{\mathcal{P}}(\text{UndefinedObject}, \texttt{\#DoIt})$$

Recall that this method exists in any valid program.

There are four elements of $\mathit{contours}_0$:

- $\mathit{contours}_0$[`NilCID`] binds the instance variables of `NilObj`. It is an empty contour.
- $\mathit{contours}_0$[`GlobalsCID`] binds the global variables. It maps each of the labels in $\mathcal{P}.\mathit{globals}$ to `NilObj`.
- $\mathit{contours}_0[\mathit{cid}_{\mathit{params}}]$ binds the parameters of the start method. Since the start method has no parameters, this contour is empty.
- $\mathit{contours}_0[\mathit{cid}_{\mathit{temps}}]$ binds the temporary variables of the start method. It maps each of the labels *startmeth.temporaries* to `NilObj`.

The attributes of $\mathit{activation}_0$ are as follows:

- *block_spec* specifies the main block of *startmeth*.
- $\mathit{pc} = 1$
- *caller* = `undef`
- *outer* = `undef`
- *receiver* = `NilObj`
- $\mathit{params\_cid} = \mathit{cid}_{\mathit{params}}$
- $\mathit{temps\_cid} = \mathit{cid}_{\mathit{temps}}$
- *caller_var* = `undef`

5.13 Execution

Execution may now be defined, given the preceding definitions. Mini-Smalltalk execution is defined as an iteration of a **step** function on the initial configuration, thus yielding a sequence of configurations. This section defines **step**. Throughout this section, let *cfg* consist of *activation* and *contours*, and let $\mathit{cfg}' = \mathbf{step}(\mathcal{P}, \mathit{cfg})$. Thus, cfg' must be defined for an arbitrary program $\mathcal{P}$ being analyzed, and an arbitrary configuration *cfg*.

Trivially, if *activation* is HALTED, then the $\mathit{cfg}' = \mathit{cfg}$. Otherwise, suppose that the *pc* of the current activation is within the bounds of its statement array. That is, let *statement* be the next statement to execute:

$$\begin{array}{rcl} \mathit{statement} & = & \mathit{block.statements}[\mathit{activation.pc}] \\ \text{where } \mathit{block} & = & \mathbf{lookup\_block}_{\mathcal{P}}(\mathit{activation.block\_spec}) \end{array}$$

and let $\mathit{activation}_{\mathit{inc}}$ be the same as *activation* except that *pc* has been incremented, i.e.

$$\mathit{activation}_{\mathit{inc}} = \mathit{activation}[\mathit{pc} \mapsto \mathit{activation.pc} + 1]$$

Let $\mathit{cfg}_{\mathit{inc}} = (\mathit{activation}_{\mathit{inc}}, \mathit{contours})$. Then there are the following cases:

- If *statement* is $[\![l := \texttt{self}]\!]$, then

$$cfg' = \textbf{write\_var}(cfg_{inc}, l, activation_{inc}.receiver)$$

- If *statement* is $[\![l := literal]\!]$, then let:

$$(litobj, contours_{lit}) = \textbf{inst\_literal}(literal, contours)$$

  Then:

$$cfg' = \textbf{write\_var}((activation_{inc}, contours_{lit}), l, litobj)$$

- If *statement* is $[\![l := l']\!]$, then let:

$$obj = \textbf{read\_var}(cfg, act, l')$$

  Then:

$$cfg' = \textbf{write\_var}(cfg_{inc}, l, obj)$$

- If *statement* is $[\![l := \texttt{new}\ \ class]\!]$, then a new object is to be created. If *class* is Block or Selector, then *cfg′* is halted; closures and selector objects cannot be created with `new` statements. Otherwise, choose *newcid* as a label not in *contours*. Let *newcontour* be a contour mapping the instance variables of *class* to `NilObj`. Let *newobject* be an object whose class is *class* and whose contour is *newcontour*. Let

$$contours_{new} = contours[newcid \mapsto newcontour]$$

  Then:

$$cfg' = \textbf{write\_var}((activation_{inc}, contours_{new}), l, newobject)$$

- If *statement* is $[\![l := block]\!]$, then let *dynblock* be a new closure whose *block_spec* is an extension of $activation_{inc}.block\_spec$ to specify the block *block*, and whose *outer* is $activation_{inc}$. Then:

$$cfg' = \textbf{write\_var}(cfg_{inc}, l, dynblock)$$

- If *statement* is $[\![l := \texttt{send}(rcvr, selector, arg_1 \ldots \text{arg}_m)]\!]$, then let:

$$\begin{aligned} rcvrobj &= \textbf{read\_var}(cfg_{inc}, act_{inc}, rcvr) \\ argobj_i &= \textbf{read\_var}(cfg_{inc}, act_{inc}, arg_i),\ \forall i \in 1 \ldots m \\ method &= \textbf{lookup}_{\mathcal{P}}(rcvrobj.class, selector) \end{aligned}$$

  If *method* is `undef`, then the method lookup failed and the machine halts. Otherwise, a new activation $activation_{called}$ is created for the called method with the following attributes:

$$\begin{aligned} block\_spec &= (method, []) \\ caller &= activation_{inc} \\ outer &= \texttt{undef} \\ receiver &= rcvrobj \\ param\_cid &= newcid_p \\ temp\_cid &= newcid_t \\ caller\_var &= l \end{aligned}$$

where $newcid_p$ and $newcid_t$ are fresh labels. Let $contour_{temps}$ be a contour mapping each of *method.temporaries* to `NilObj`, and let $contour_{params}$ be a contour mapping $method.parameters_i$ to `NilObj` for each $i \in 1 \dots m$. Let $contours_{called}$ be *contours* with these two contours added:

$$contours_{called} = \\ contours[newcid_p \mapsto contour_{params},\ newcid_t \mapsto contour_{temps}]$$

The final configuration is then:

$$cfg' = (activation_{called}, contours_{called})$$

- If *statement* is $[\![l := \texttt{sendvar}(rcvr, l_{sel}, arg_1 \dots \mathrm{arg}_m)]\!]$, then cfg' is computed as if the statement were a `send` statement, with the exception that the method selector is:

  $$\textbf{read\_var}(cfg_{inc}, activation_{inc}, l_{sel})$$

  If the selector is not a selector object, then cfg' is halted. Otherwise, cfg' is as described for `send` statements.

- If *statement* is $[\![l := \texttt{beval}(l_b, arg_1 \dots \mathrm{arg}_m)]\!]$, then cfg' is similar to that resulting from a `send` statement. Let:

  $$dynblock = \textbf{read\_var}(cfg_{inc}, act_{inc}, l_b)$$

  If *dynblock* is not actually a block, or if the number of arguments supplied is different than *dynblock* requires, then cfg' is halted. Otherwise, look up the arguments, just as with a message send:

  $$argobj_i = \textbf{read\_var}(cfg_{inc}, act_{inc}, arg_i),\ \forall i \in 1 \dots m$$

  Create new labels $newcid_p$ and $newcid_t$, and new contours $contour_{params}$ and $contour_{temps}$, just as with a `send` statement. The new activation, *activation′*, will then have the following attributes:

  $$\begin{array}{rcl} block\_spec &=& dynblock \\ pc &=& 1 \\ contourid &=& newcid \\ caller &=& activation_{inc} \\ outer &=& blockvar.outer \\ receiver &=& blockvar.outer.receiver \\ params\_cid &=& newcid_p \\ temps\_cid &=& newcid_t \\ caller\_var &=& lvar \end{array}$$

  Then:

  $$cfg' = (activation', contours[newcid \mapsto contour_{called}], heap, globals)$$

Finally, suppose *pc* is larger than the number of statements in the current activation. The current block will return some value. Let l_{ret} be the name of the variable that is to be returned. There are two cases:

- Suppose the block returns values from the surrounding method (*retFromMethod* is true). Then let:

  $$callact = \textbf{outermost}(activation).caller$$

If *callact* is `undef` then execution halts. Otherwise, look up the object to return:

$$retobj = \textbf{read\_var}(cfg_{inc}, act_{inc}, l_{ret})$$

and write the appropriate variable and return to the calling activation:

$$cfg' = \textbf{write\_var}((callact, contours), cfg_{inc}.caller\_var, retobj)$$

- Suppose the block returns values from the current block (*retFromMethod* is false). Let:

  $$retobj = \textbf{read\_var}(cfg_{inc}, act_{inc}, l_{ret})$$

  Then:

  $$cfg' = \textbf{write\_var}((act_{inc}.caller, contours), act_{inc}.outer\_var, retobj)$$

5.14 Various semantic properties

Lemma 5.1 (Semantic Sanity)**.** Mini-Smalltalk semantics has many of the properties one would expect. Several properties are listed below. For each of these properties, $\mathcal{P}$ is any program, n is any non-negative integer, $act \in \textbf{all\_activations}(\textbf{step}_n(\mathcal{P}))$, and $obj \in \textbf{all\_objects}(\textbf{step}_n(\mathcal{P}))$.

- Only the initial activation has a *caller* or a *caller_var* that is `undef`. In particular:

  $$act.caller = \texttt{undef} \quad \Leftrightarrow \quad act.caller\_var = \texttt{undef}$$

- The *pc* of an activation either points within the range of its available statements or points one past the end:

  $$1 \;\leq\; act.pc \;\leq\; \textbf{len}(\textbf{lookup\_block}(act.block\_spec).statements) + 1$$

- All contours are within the *contours* of the configuration:

  $$\begin{aligned} obj.cid &\in \textbf{domain}(\textbf{step}_n(\mathcal{P}).contours) \\ act.params\_cid &\in \textbf{domain}(\textbf{step}_n(\mathcal{P}).contours) \\ act.temps\_cid &\in \textbf{domain}(\textbf{step}_n(\mathcal{P}).contours) \end{aligned}$$

- The domain of $\textbf{step}_n(\mathcal{P}).contours[obj.cid]$ is precisely the set of instance variables of the object's class, *obj.class*.

- The class of *act.receiver* is either the class named *act.block_spec.class_name* or a descendent of that class.

- If $act \in \textbf{all\_activations}(\textbf{step}_n(\mathcal{P}))$ is an activation for a block other than a method's main block, then *act.caller* is an activation whose $(pc - 1)$th statement is a `beval` statement. Likewise, if *act* is an activation for a method's main block, then the caller's $(pc - 1)$th statement is a `send` or `sendvar` statement.

- If $act.outer \neq \texttt{undef}$, then $act.outer.receiver = act.receiver$.

- If *act.block_spec* is the main block of a method, then $act.outer = \texttt{undef}$.

- If *act.block_spec* is not the main block of a method, then *act.outer.block_spec* is the block immediately enclosing *act.block_spec*.

Proof. The proof is straightforward by induction on the number of execution steps. □

The following lemma claims that contour ids are unique, with only one class of exceptions. The contour id used to reference an object's instance variables, for example, is never used by a different object and never used by an activation to refer to parameters or temporary variables. The contour id used by one activation is never used by another, *except* that activations differing only by their *pc* are considered the same activation at different stages of execution. This unfortunately complicated exception allows activations themselves to be immutable, thus simplifying other parts of the semantics. There is no need, for example, for an *activation id*.

Lemma 5.2 (Distinct Contours)**.** In a given configuration, there is no contour id of an object that is also a contour id for an activation. There is no contour id for two different objects. There is no contour id for two activations that differ by more than their *pc*'s.

Proof. The proof is straightforward by induction on the number of execution steps. Note that whenever the **step** creates a new object or activation, it uses fresh contour id's. □

Lemma 5.3 (Send History for Methods)**.** Suppose that $\mathbf{step}_n(\mathcal{P})$ is not halted and that

$$\begin{aligned} act &\in \mathbf{all\_activations}(\mathbf{step}_n(\mathcal{P})) \\ act.outermost.sender &\neq \texttt{undef} \end{aligned}$$

Then there is an $m < n$ and an i such that under these definitions:

$$\begin{aligned} method &= \mathbf{lookup\_meth}(act.outermost.block) \\ selector &= method.selector \\ cfg_m &= \mathbf{step}_m(\mathcal{P}) = (act_m, cnt_m) \\ statement_{mi} &= \mathbf{lookup\_block}(act_m.block).statements(i) \end{aligned}$$

$statement_{mi}$ is either a `send` statement with selector *selector*, or a `sendvar` statement. If $statement_{mi}$ is a `sendvar` statement, then it reads its selector from some variable *selectorvar* such that:

$$sobj = \mathbf{read\_var}(cfg_m, act_m, selectorvar)$$

where *sobj* is a selector object for *selector*. The variable assigned by $statement_{mi}$ is the variable recorded in *act.outermost.caller_var*. Furthermore, the receiver *rcvr* of the `send` statement is such that:

$$\mathbf{lookup}_{\mathcal{P}}(\mathbf{read\_var}(cfg_m, act_m, rcvr).class, selector) = method_n$$

and each parameter has the value specified in the call statement:

$$\forall k : \; act.outermost.params_k = \mathbf{read\_var}(cfg_m, act_m, argvar_k)$$

Proof. The proof is by induction on the number of execution steps. The lemma is trivially correct for the initial configuration. Suppose it is true for $\mathbf{step}_{n-1}(\mathcal{P})$, and it will be shown for $\mathbf{step}_n(\mathcal{P})$.

If the next statement to execute in configuration $n-1$ is a `send` or `sendvar` statement, then all activations in $\mathbf{step}_n(\mathcal{P})$ but one are also activations in $\mathbf{step}_n(\mathcal{P})$, disregarding changes to *pc*'s. For the solitary new activation, choose $m = n - 1$ and i as the current pc from $\mathbf{step}_{n-1}(\mathcal{P})$, and the conditions will clearly be true. For all other activations, choose the same m and i as was chosen for each activation in $\mathbf{step}_{n-1}(\mathcal{P})$.

If the next statement is a `beval`, then again there is only one new activation. For that activation, choose the same m and i as was chosen for its *outer* activation. For the other activations, choose the same m and i as before.

If the next statement is not a `send`, `sendvar`, or `beval` statement, then the new activations are a subset of the old ones, and the same m's and i's may be chosen for step n as for step $n-1$. Note that no statement in Mini-Smalltalk may bind a parameter to a different object in an existing activation; the only way to bind a parameter is to create a new activation. □

Lemma 5.4 (Send History for Blocks)**.** Suppose that $\mathbf{step}_n(\mathcal{P})$ is not halted and that

$$\begin{aligned} act &\in \mathbf{all\_activations}(\mathbf{step}_n(\mathcal{P})) \\ act.outer &\neq \texttt{undef} \end{aligned}$$

Then there is an $m < n$ and an i such that under these definitions:

$$\begin{aligned} cfg_m &= \mathbf{step}_m(\mathcal{P}) = (act_m, cnt_m) \\ statement_{mi} &= act_m.block.statements(i) \end{aligned}$$

$statement_{mi}$ is a `beval` statement. The variable `blockvar` that it reads its variable from is such that:

$$blockobj = \mathbf{read\_var}(cfg_m, act_m, blockvar)$$

where *blockobj* is a closure for block *block* and outer activation *act.outer*. The variable assigned by $statement_{mi}$ is *act.caller_var*. Finally, each parameter has the value specified in the `beval` statement:

$$\forall k: \ act.params_k = \mathbf{read\_var}(cfg_m, act_m, argvar_k)$$

Proof. The proof is similar to that for the Send History Lemma for Methods. Induct on the number of execution steps. The lemma is trivially true for the initial configuration. Suppose that it is true in configuration $\mathbf{step}_{n-1}(\mathcal{P})$, and it will be shown that it is also true in $\mathbf{step}_n(\mathcal{P})$.

If the next statement to execute in $\mathbf{step}_{n-1}(\mathcal{P})$ is a `beval` statement, then there is one new activation in $\mathbf{step}_n(\mathcal{P})$. If *act* is that activation, choose the same m and i as was chosen for its *outer* activation. For the other activations, choose the same m and i as before.

If the next statement is a `send` or `sendvar` statement, then there is only one new activation, and *act* cannot be that one because $act.outer = \texttt{undef}$. For all other activations, choose the same m and i as in $\mathbf{step}_{n-1}(\mathcal{P})$.

If the next statement is neither a `beval`, `send`, nor `sendvar` statement, then choose the same m and i for *act* as was chosen in $\mathbf{step}_{n-1}(\mathcal{P})$. □

Chapter 6

Data-flow analysis in Mini-Smalltalk

This chapter continues the formal description of **DDP** by describing a general framework for discussing data flow in Mini-Smalltalk. Thus, this chapter formally describes the *answer* provided by a type inference execution, as well as the intermediate structures a type inferencer uses while it runs.

6.1 Preliminaries

The function **remove_redundancies** operates on any set with a relation $\sqsubseteq$. It removes all elements from a set that are subsumed by another element of the set. Formally, it is defined as follows:

$$\textbf{remove\_redundancies}(s) = \{x \in s \mid \neg\exists y \in s : x \neq y \wedge x \sqsubseteq y\}$$

6.2 Variables

The result of a type inference and the rules of justification for those results are defined in terms of the static program and its variables; they are statements such as “this variable has this type”. The correctness of those results is defined in terms of the dynamic behavior of the program; e.g. “this variable has this type, in this configuration”. Yet, the semantics of the program are given in terms of labels, not in terms of any concept of “variable”. Thus there is a disconnect between how the algorithm results are stated, how the correctness criteria are stated, and how the semantics is stated. This disconnect is bridged by *variables*.

6.2.1 Definition

Figure 6.1 gives a summary of the four possible kinds of variables in Mini-Smalltalk. The meaning of each kind should be apparent from its name.

$$\begin{array}{rcl}
\langle \textit{variable} \rangle & ::= & \text{GlobalVar named:} \langle \textit{label} \rangle \\
 & \mid & \text{InstanceVar ofClass:} \langle \textit{label} \rangle \ \ \text{named:} \langle \textit{label} \rangle \\
 & \mid & \text{Parameter ofBlock:} \langle \textit{block\_spec} \rangle \ \ \text{named:} \langle \textit{label} \rangle \\
 & \mid & \text{TemporaryVar ofBlock:} \langle \textit{block\_spec} \rangle \ \ \text{named:} \langle \textit{label} \rangle
\end{array}$$

Figure 6.1: Variables

6.2.2 Variables found dynamically

The semantics has been carefully defined so that every activation is linked to the appropriate block from the original program. Thus, every variable reference that occurs during execution may be traced to the associated static variable from the program. To do this, an analyzer begins at an activation and looks at the temporaries and then the parameters of the activation; if there are any outer activations, then their temporaries and parameters are checked as well; if the label appears in none of these activations, then the instance variables are checked. Finally, if none of these locations binds the variable, the global variables are checked.

The function **dynamic_bindings**, defined in Figure 6.2, performs this search. Given any configuration and an activation within that configuration, **dynamic_bindings** will find a *binding map* describing the variables readable from *act*. A binding map is a partial function whose domain includes a finite number of labels plus the special values `method` and `block`. The binding map maps each readable label to a variable describing the variable that will be read from if that label is read from the specified activation. A binding map also maps `method` and `block` to the method and block that are executing.

6.2.3 Variables found statically

Since Mini-Smalltalk uses lexically bound variables, static analysis can predict which variables will be bound by each variable reference in the program. The function **static_bindings**, defined in Figure 6.3, maps a block specification to a binding map. It finds variable bindings by tracing through blocks, then class definitions, and finally the list of globals declared in the program.

A *valid variable* for a program $\mathcal{P}$ is one that is in the static bindings of some block of the program, i.e.:

$$\exists block\_spec : \exists l : \quad var = \mathbf{static\_bindings}_{\mathcal{P}}(block\_spec)[l]$$

The function **bound_stats** may be used to enumerate the statements of a program along with variable information. **bound_stats**($\mathcal{P}$) is the set of all statements in the program paired with the binding maps that are in effect for those statements. Formally, it is defined as the smallest set satisfying:

$$\frac{\begin{array}{c} bs \in \mathbf{all\_blocks}(\mathcal{P}) \\ stat \in \mathbf{block}_{\mathcal{P}}(bs).statements \\ bindings = \mathbf{static\_bindings}_{\mathcal{P}}(bs) \end{array}}{(stat, bindings) \in \mathbf{bound\_stats}(\mathcal{P})}$$

6.2.4 Properties of variables

This section proves a few useful properties about variables.

Theorem 1 (Lexical Binding of Mini-Smalltalk)**.** *For any program $\mathcal{P}$, for any configuration $cfg = \mathbf{step}_n(\mathcal{P})$, and for any activation $act \in \mathbf{all\_activations}(cfg)$:*

$$\mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act) = \mathbf{static\_bindings}_{\mathcal{P}}(act.block\_spec)$$

Proof. The proof is by induction on the number of steps of execution.

In configuration $\mathbf{step}_0(\mathcal{P})$, the property is straightforward to show by a case analysis. Consider, in turn, labels for the temporary variables of the start method, the parameters of the start method, the global variables, and labels that are none of these. The static and dynamic binding of the labels are the same in each case.

Suppose then that the property is true in $\mathbf{step}_n(\mathcal{P})$; it must be shown that it is still true in $\mathbf{step}_{n+1}(\mathcal{P})$. To avoid triviality, suppose that neither configuration is halted.

If $\mathbf{step}_n(\mathcal{P})$ executes a `send` or `sendvar` statement to reach $\mathbf{step}_{n+1}(\mathcal{P})$, then there is precisely one new activation in $\mathbf{step}_{n+1}(\mathcal{P})$, disregarding changes to *pc*'s. As with the argument in the initial configuration, it is straightforward to show that the property holds in this new activation.

$$\frac{\begin{array}{c}
act.outer = \texttt{undef} \\
block\_spec = act.block\_spec \\
meth\_spec = block\_spec.meth\_spec \qquad g_1 \ldots g_p = \mathbf{domain}(cnt[\texttt{GlobalsCID}]) \\
\forall\ k \in 1 \ldots p:\ gv_k = (\text{GlobalVariable named: } g_k) \qquad i_1 \ldots i_q = \mathbf{domain}(cnt[act.rcvr.ivars\_cid]) \\
\forall\ k \in 1 \ldots q:\ iv_k = (\text{InstanceVariable ofClass: } act.rcvr.class \text{ named: } i_k) \\
p_1 \ldots p_r = \mathbf{domain}(cnt[act.params\_cid]) \\
\forall\ k \in 1 \ldots r:\ pv_k = (\text{Parameter ofBlock: } block\_spec \text{ named: } p_k) \\
t_1 \ldots t_s = \mathbf{domain}(cnt[act.temps\_cid]) \\
\forall\ k \in 1 \ldots s:\ tv_k = (\text{TemporaryVar ofBlock: } block\_spec \text{ named: } t_k) \\
bindings_0 = [\texttt{method} \rightarrow meth\_spec,\ \texttt{block} \rightarrow block\_spec] \\
bindings_1 = bindings_0[g_1 \rightarrow gv_1,\ \ldots,\ g_p \rightarrow gv_p] \\
bindings_2 = bindings_1[i_1 \rightarrow iv_1,\ \ldots,\ i_q \rightarrow iv_q] \\
bindings_3 = bindings_2[p_1 \rightarrow pv_1,\ \ldots,\ p_r \rightarrow pv_r] \\
bindings = bindings_3[t_1 \rightarrow tv_1,\ \ldots,\ t_s \rightarrow tv_s]
\end{array}}{bindings = \mathbf{dynamic\_bindings}_{\mathcal{P}}((topact, cnt), act)}$$

$$\frac{\begin{array}{c}
act.outer \neq \texttt{undef} \\
block\_spec = act.block\_spec \qquad p_1 \ldots p_r = \mathbf{domain}(cnt[act.params\_cid]) \\
\forall\ k \in 1 \ldots r:\ pv_k = (\text{Parameter ofBlock: } block\_spec \text{ named: } p_k) \\
t_1 \ldots t_s = \mathbf{domain}(cnt[act.temps\_cid]) \\
\forall\ k \in 1 \ldots s:\ tv_k = (\text{TemporaryVar ofBlock: } block\_spec \text{ named: } t_k) \\
bindings_{outer} = \mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act.outer) \qquad bindings_0 = bindings_{outer}[\texttt{block} \rightarrow block\_spec] \\
bindings_1 = bindings_0[p_1 \rightarrow pv_1,\ \ldots,\ p_r \rightarrow pv_r] \\
bindings = bindings_1[t_1 \rightarrow tv_1,\ \ldots,\ t_s \rightarrow tv_s]
\end{array}}{bindings = \mathbf{dynamic\_bindings}_{\mathcal{P}}((topact, cnt), act)}$$

Figure 6.2: Dynamic Variable Binding

$$\frac{\begin{array}{c}
\mathit{block\_spec} = \text{BlockSpec methodSpec: } \mathit{method\_spec} \ \text{ statementNums: []} \\
\mathit{method\_spec} = \text{MethodSpec className: } \mathit{class} \ \text{ selector: } \mathit{selector} \\
\mathit{method} = \mathbf{lookup\_meth}_{\mathcal{P}}(\mathit{method\_spec}) \\
\mathit{method} = \text{Method block: } \mathit{block} \\
\mathit{block} = \text{Block parameters: } \mathit{params} \ \text{ temporaries: } \mathit{temps} \ \text{ statements: } \mathit{stats} \\
\mathcal{P} = \text{Program globals: } \mathit{globals} \ \text{ classes: } \mathit{classes} \\
g_1 \ldots g_p = \mathit{globals} \qquad \forall\, k \in 1 \ldots p : \ gv_k = (\text{GlobalVariable named: } g_k) \\
i_1 \ldots i_q = \mathbf{all\_instvars}_{\mathcal{P}}(\mathit{class}) \qquad \forall\, k \in 1 \ldots q : \ iv_k = (\text{InstanceVariable ofClass: } \mathit{class} \ \text{ named: } i_k) \\
p_1 \ldots p_r = \mathit{params} \qquad \forall\, k \in 1 \ldots r : \ pv_k = (\text{Parameter ofBlock: } \mathit{block\_spec} \ \text{ named: } p_k) \\
t_1 \ldots t_s = \mathit{temps} \qquad \forall\, k \in 1 \ldots s : \ tv_k = (\text{TemporaryVar ofBlock: } \mathit{block\_spec} \ \text{ named: } t_k) \\
\mathit{bindings}_0 = [\texttt{method} \to \mathit{meth\_spec},\ \texttt{block} \to \mathit{block\_spec}] \\
\mathit{bindings}_1 = \mathit{bindings}_0[g_1 \to gv_1,\ \ldots,\ g_p \to gv_p] \\
\mathit{bindings}_2 = \mathit{bindings}_1[i_1 \to iv_1,\ \ldots,\ i_q \to iv_q] \\
\mathit{bindings}_3 = \mathit{bindings}_2[p_1 \to pv_1,\ \ldots,\ p_r \to pv_r] \\
\mathit{bindings} = \mathit{bindings}_3[t_1 \to tv_1,\ \ldots,\ t_s \to tv_s]
\end{array}}{\mathit{bindings} = \mathbf{static\_bindings}_{\mathcal{P}}(\mathit{block\_spec})}$$

$$\frac{\begin{array}{c}
\mathit{block\_spec} = \text{BlockSpec methodSpec: } \mathit{method\_spec} \ \text{ statementNums: } \mathit{snums} \\
\mathit{snums} = \mathbf{append}(\mathit{snums}', \mathit{snum}) \\
\mathit{block} = \mathbf{lookup\_block}_{\mathcal{P}}(\mathit{block\_spec}) \\
\mathit{block\_spec}' = \text{BlockSpec methodSpec: } \mathit{method\_spec} \ \text{ statementNums: } \mathit{snums}' \\
\mathit{block} = \text{Block parameters: } \mathit{params} \ \text{ temporaries: } \mathit{temps} \ \text{ statements: } \mathit{stats} \\
p_1 \ldots p_r = \mathit{params} \qquad \forall\, k \in 1 \ldots r : \ pv_k = (\text{Parameter ofBlock: } \mathit{block\_spec} \ \text{ named: } p_k) \\
t_1 \ldots t_s = \mathit{temps} \qquad \forall\, k \in 1 \ldots s : \ tv_k = (\text{TemporaryVar ofBlock: } \mathit{block\_spec} \ \text{ named: } t_k) \\
\mathit{bindings}_0 = \mathbf{static\_bindings}_{\mathcal{P}}(\mathit{block\_spec}') \qquad \mathit{bindings}_1 = \mathit{bindings}_0[p_1 \to pv_1,\ \ldots,\ p_r \to pv_r] \\
\mathit{bindings}_2 = \mathit{bindings}_1[t_1 \to tv_1,\ \ldots,\ t_s \to tv_s] \\
\mathit{bindings} = \mathit{bindings}_2[\texttt{block} \to \mathit{block\_spec}]
\end{array}}{\mathit{bindings} = \mathbf{static\_bindings}_{\mathcal{P}}(\mathit{block\_spec})}$$

Figure 6.3: Static Variable Binding

Suppose then that $\mathbf{step}_n(\mathcal{P})$ executes a `beval` statement. Again, there is one new activation, but now the new activation has an *outer* activation. Consider any label l. If l is a temporary variable or parameter of the new activation, then it is straightforward to show that the static and dynamic bindings are the same. Otherwise, the dynamic binding of l in the new activation is the same as the dynamic binding of l in the new activation's outer activation. Further, the static binding of l in the new activation's block is the same as the static binding of l in the activation surrounding the new activation's block. By the Semantic Sanity Lemma, the block of the outer activation, must be the same as the outer block of the new activation. Thus, by the inductive assumption, the static and dynamic bindings of the new activation must be the same.

If $\mathbf{step}_n(\mathcal{P})$ executes some other statement, or returns from a block, then there are no new activations in $\mathbf{step}_{n+1}(\mathcal{P})$.

Thus in all cases, the property remains true in $\mathbf{step}_{n+1}(\mathcal{P})$. □

The following lemma shows that the same contour is never used to hold different variables.

Lemma 6.1 (Unshared Contours)**.** Suppose that $cfg = \mathbf{step}_n(\mathcal{P})$, that act_1 and act_2 are among $\mathbf{all\_activations}(cfg)$, and that l is any label for a variable readable in both act_1 and act_2. Then:

$$\begin{aligned}&\mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_1, l, \texttt{false})\\ &\quad = \mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_2, l, \texttt{false})\\ &\quad\quad \Rightarrow \mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act_1)[l]\\ &\quad\quad\quad = \mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act_2)[l]\end{aligned}$$

Proof. The proof is by induction on the number of steps of program execution. In the initial configuration, there is only one activation, and the proof is trivial. Suppose, then, that the statement is true in $\mathbf{step}_n(\mathcal{P})$; let us show that it is true in $\mathbf{step}_{n+1}(\mathcal{P})$. Assume, to avoid triviality, that $\mathbf{step}_{n+1}(\mathcal{P})$ is not halted.

If the statement to execute is a `send` or `sendvar` statement, then there is one new activation in $\mathbf{step}_{n+1}(\mathcal{P})$ that was not present in $\mathbf{step}_n(\mathcal{P})$. Suppose act_1 is the newly created activation, and act_2 is some other activation. The activation act_1 has a newly created contour, and $\mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_1, l, \texttt{false})$ must be that contour because act_1 has no outer activation. On the other hand, the contour $\mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_2, l, \texttt{false})$ must have existed in $\mathbf{step}_n(\mathcal{P})$. Thus, the two contours cannot be the same, and the desired statement is vacuously true. Likewise if act_2 is the newly created contour. If $act_1 = act_2$ then the proof is trivial. If neither act_1 nor act_2 is the newly created activation, then the proof is by the inductive assumption.

If the statement to execute is a `beval` statement, then again there is one new activation created and one new contour. Suppose that act_1 is the newly created activation; the other cases need no further attention. If $\mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_1, l, \texttt{false})$ is the newly created contour, then the statement is vacuously true. If it is some other contour, then it must be the same as $\mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act_1.outer, l, \texttt{false})$. Thus,

$$\mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act_1)[l] = \mathbf{dynamic\_bindings}_{\mathcal{P}}(cfg, act_1.outer)[l]$$

Since $act_1.outer$ is an activation that was present in $\mathbf{step}_n(\mathcal{P})$, the inductive assumption gives the desired property.

All other statement types do not create any new activations, and thus the inductive assumption is already strong enough to give the desired property. □

6.3 Types

A *type* is a set of objects. Types in **DDP** must be in one of the following forms:

- $\lfloor C \rceil$ is the *class type* containing all objects whose class is named C.

- $S\llparenthesis s, m\rrparenthesis$ is the *selector type* containing selector objects whose label is s and whose number of arguments is m.

- $B\llparenthesis bs\rrparenthesis_{ctx}$ is the *block type* containing all closures created by the statement specified by bs, whose outer activation matches context ctx. Contexts are defined in the next section; types and contexts are defined with mutual recursion. ctx must not be $\bot_{ctx}$, the empty context.

- Σts is a *sum type* where ts is a finite set of the above kinds of types. It includes all objects that are included in any of the elements of ts. No element of ts may be a subtype of another. The size of ts must be at least 2.

- $\bot$ is the *empty type*, the type including no objects.

- $\top$ is the *universal type*, the type including all objects.

Additionally, the notation $\llparenthesis C\rrparenthesis^{+}$ is shorthand for a "class cone type" which includes all objects that are members of C or a subclass of C. It is only well-defined in the context of an implicitly understood program $\mathcal{P}$. Formally,

$$\llparenthesis C\rrparenthesis^{+} = \Sigma\{\llparenthesis C'\rrparenthesis \mid C' = C \quad \vee \quad C' \text{ inherits from } C\}$$

Notice that subtyping is separated from subclassing in **DDP**. The type $\llparenthesis$Integer$\rrparenthesis$ is *not* a subtype of $\llparenthesis$Number$\rrparenthesis$. The type $\llparenthesis$Number$\rrparenthesis$ includes only those objects whose class is exactly Number, not whose class is Number or a subclass of Number. In other words, $\llparenthesis$Number$\rrparenthesis$ and $\llparenthesis$Number$\rrparenthesis^{+}$ are different types.

Types may be compared with the *subtype* relationship defined in Figure 6.4. The relation is defined such that whenever $t_1 \sqsubseteq t_2$ and *object* is a member of t_1, then *object* is also a member of t_2. Additionally, types may be combined using the $\sqcup$ and $\sqcap$ relations defined in Figure 6.5 and Figure 6.6 respectively. As section 6.10 shows, these relations define proper join and meet operations. Further, a review of the definitions is enough to see that if an object is in both t_1 and t_2, then it is also in $t_1 \sqcap t_2$.

The function **lit_type** returns a type for a literal. Its details are left unspecified, but **lit_type** must be compatible with **inst_literal**: the object created by **inst_literal**(*lit*) must be an element of type **lit_type**(*lit*).

lookup$^{*}_{\mathcal{P}}(type, selector)$ is the set of methods that may respond if *selector* is sent to an object of type *type*. It is the set containing, for each class *class* of any object in *type*, the method **lookup**$_{\mathcal{P}}(class, selector)$.

6.4 Dynamic context

In general, better results can be obtained for the type of a message-send expression if the responding methods are analyzed multiple times, once for each possible combination of argument types. Such a combination of argument types form a *context*. Formally, a context is a function: $ctx(act, cfg)$ is true whenever the context ctx matches the activation *act* that is part of configuration *cfg*.

The largest context used in this paper is $\top_{ctx}$, a context matching any activation. The smallest context is $\bot_{ctx}$, a context matching no activation.

The only non-trivial kind of context used in this paper is a *parameters context*. A parameters context specifies a block, a type for the method receiver, and a complete function from parameter variables to types. This function must map a finite number of parameters into types other than $\top$. A parameters context is written like this:

$$< (bs)\ \texttt{self} : \llparenthesis\text{SmallInteger}\rrparenthesis, \texttt{anInteger} : \llparenthesis\text{LargePositiveInteger}\rrparenthesis >$$

This context is for block bs, which must be neither $\top_{bs}$ nor $\bot_{bs}$. It assigns a type of $\llparenthesis$SmallInteger$\rrparenthesis$ to the method receiver, and it assigns a type of $\llparenthesis$LargePositiveInteger$\rrparenthesis$ to the `anInteger` parameter. It assigns a type of $\top$ to all other parameters.

A parameters context matches an activation in the expected way: the activation must be for a block that is lexically within the specified block, the activation must have a receiver type that is a member of the specified

TO-REFL
$$\frac{}{t \sqsubseteq t}$$

TO-TOP
$$\frac{}{t \sqsubseteq \top}$$

TO-BOTTOM
$$\frac{}{\bot \sqsubseteq t}$$

TO-BLOCK-CTX
$$\frac{ctx_1 \sqsubseteq ctx_2}{B\{\!|bs|\!\}_{ctx_1} \sqsubseteq B\{\!|bs|\!\}_{ctx_2}}$$

TO-BLOCK-CLASS
$$\frac{}{B\{\!|bs|\!\}_{ctx} \sqsubseteq \{\!|\text{Block}|\!\}}$$

TO-SELECTOR
$$\frac{}{S\{\!|s,m|\!\} \sqsubseteq \{\!|\text{Selector}|\!\}}$$

TO-SUM-R
$$\frac{t' \in ts \qquad t \sqsubseteq t'}{t \sqsubseteq \Sigma ts}$$

TO-SUM-L
$$\frac{\forall t' \in ts:\ t' \sqsubseteq t}{\Sigma ts \sqsubseteq t}$$

Figure 6.4: Subtyping

TJ-SUB1
$$\frac{t_1 \sqsubseteq t_2}{t_1 \sqcup t_2 = t_2}$$

TJ-SUB2
$$\frac{t_2 \sqsubseteq t_1}{t_1 \sqcup t_2 = t_1}$$

TJ-SIMPLES
$$\frac{\begin{array}{c} t_1 \not\sqsubseteq t_2 \qquad t_2 \not\sqsubseteq t_1 \\ t_1 \textit{ is a class, selector, or block type} \\ t_2 \textit{ is a class, selector, or block type} \end{array}}{t_1 \sqcup t_2 = \Sigma\{t_1, t_2\}}$$

TJ-MIXED1
$$\frac{\begin{array}{c} t_1 \textit{ is a class, selector, or block type} \\ t_2 = \Sigma ts \qquad t_1 \not\sqsubseteq t_2 \qquad t_2 \not\sqsubseteq t_1 \\ ts' = \textbf{remove\_redundancies}(ts \cup \{t_1\}) \end{array}}{t_1 \sqcup t_2 = \Sigma ts'}$$

TJ-MIXED2
$$\frac{\begin{array}{c} t_2 \textit{ is a class, selector, or block type} \\ t_1 = \Sigma ts \qquad t_1 \not\sqsubseteq t_2 \qquad t_2 \not\sqsubseteq t_1 \\ ts' = \textbf{remove\_redundancies}(ts \cup \{t_2\}) \end{array}}{t_1 \sqcup t_2 = \Sigma ts'}$$

TJ-SUMS
$$\frac{\begin{array}{c} t_1 = \Sigma ts_1 \qquad t_2 = \Sigma ts_2 \\ ts = \textbf{remove\_redundecies}(ts_1 \cup ts_2) \qquad t_1 \not\sqsubseteq t_2 \qquad t_2 \not\sqsubseteq t_1 \end{array}}{t_1 \sqcup t_2 = \Sigma ts}$$

Figure 6.5: Join for Types

TM-SYM
$$\frac{t_1 \sqcap t_2 = t_3}{t_2 \sqcap t_1 = t_3}$$

TM-SUBTYPE
$$\frac{t_1 \sqsubseteq t_2}{t_1 \sqcap t_2 = t_1}$$

TM-CLASS
$$\frac{C_1 \neq C_2}{\{\!|C_1|\!\} \sqcap \{\!|C_2|\!\} = \bot}$$

TM-CLASS-SELECTOR
$$\frac{C \neq \text{Selector}}{\{\!|C|\!\} \sqcap S\{\!|s,m|\!\} = \bot}$$

TM-CLASS-BLOCK
$$\frac{C \neq \text{Block}}{\{\!|C|\!\} \sqcap B\{\!|bs|\!\}_{ctx} = \bot}$$

TM-SELECTOR1
$$\frac{s_1 \neq s_2}{S\{\!|s_1,m_1|\!\} \sqcap S\{\!|s_2,m_2|\!\} = \bot}$$

TM-SELECTOR2
$$\frac{m_1 \neq m_2}{S\{\!|s_1,m_1|\!\} \sqcap S\{\!|s_2,m_2|\!\} = \bot}$$

TM-SEL-BLOCK
$$\frac{}{S\{\!|s,m|\!\} \sqcap B\{\!|bs|\!\}_{ctx} = \bot}$$

TM-BLOCK-DIFF
$$\frac{bs_1 \neq bs_2}{B\{\!|bs_1|\!\}_{ctx_1} \sqcap B\{\!|bs_2|\!\}_{ctx_2} = \bot}$$

TM-BLOCK-SAME
$$\frac{ctx_1 \sqcap ctx_2 = ctx}{B\{\!|bs|\!\}_{ctx_1} \sqcap B\{\!|bs|\!\}_{ctx_2} = B\{\!|bs|\!\}_{ctx}}$$

TM-SUM
$$\frac{\begin{array}{c} \forall t' \in ts : t' \sqcap t_2 = m(t') \\ t_3 = \bigsqcup_{t' \in ts} m(t') \end{array}}{\Sigma ts \sqcap t_2 = t_3}$$

Figure 6.6: Meet for Types

type, and each parameter in the activation—including those in lexically nested scopes—must hold an object that is a member of the type specified by the activation.

Formally, the attributes of a parameters context are *ctx.bs*, *ctx.selftype*, and *ctx.paramtypes*. For shorthand, however, *ctx*[`self`] refers to the type *ctx* assigns to the receiver. Likewise, *ctx*[*param*] refers to the type assigned to *param*.

Contexts may be compared to each other using the rules in Figure 6.7. Whenever $ctx_1 \sqsubseteq ctx_2$, ctx_1 matches a subset of the activations that ctx_2 matches. Two contexts may also be combined or intersected, according to the rules in Figure 6.8 and Figure 6.9. It is proven in section 6.10 that $\sqcup$ and $\sqcap$ define proper join and meet operations.

Note that when contexts for unrelated blocks are combined with $\sqcup$, the resulting context is $\top_{ctx}$. It would be possible to enrich the definition of contexts—by adding "sum contexts" as an analog to sum types—but since the present analysis never considers such unions, the added complexity would not be helpful. Intersection via $\sqcap$, on the other hand, does match precisely the contexts matched by both of two contexts that are intersected.

There are restrictions on the contexts actually used by **DDP**; see section 6.9 below for details.

6.5 Flow positions

A *flow position* describes locations that an object might be bound during program execution. It is one of the following:

1. $[\colon V\ var \colon]_{ctx}$, a *variable flow position*, describing the variable *var* in context *ctx*. *ctx* may not be $\bot_{ctx}$.
2. $[\colon S\ \ meth \colon]_{ctx}$, a *self flow position*, describing the receiver of the method *meth* executing in context *ctx*. *ctx* may not be $\bot_{ctx}$.
3. $[\colon \Sigma\, fs \colon]$, where *fs* is a finite set of flow positions of the above kinds, is a *sum flow position*. No element of *fs* may be subsumed by another. The size of *ts* must be at least 2.
4. $\top_{fp}$, the *universal flow position*, which includes all possible flow positions.
5. $\bot_{fp}$, the *empty flow position*, which includes no flow positions.

Some flow positions are completely *subsumed* by other flow positions. The rules for deciding are given in Figure 6.10. Further, flow positions may be combined with the rules in Figure 6.11 and Figure 6.12. It is proven in section 6.10 that $\sqcup$ and $\sqcap$ define proper join and meet operations for the lattice of flow positions.

An object *object* is *included in a flow position f* in configuration *cfg*, where

$$cfg = (act, cnt) = \mathbf{step}_n(\mathcal{P})$$

if all of the following are true:

1. For all global variables *var* that are valid for $\mathcal{P}$,

$$\begin{aligned} & cnt[\texttt{GlobalsCID}][var.label] = object \\ \Rightarrow \quad & [\colon V\ var \colon]_{\top_{ctx}} \sqsubseteq f \end{aligned}$$

2. For all valid instance variables *var* and valid objects *object′* whose class is a subclass of *var.class*,

$$\begin{aligned} & cnt[object'.ivars\_cid][var.label] = object \\ \Rightarrow \quad & [\colon V\ var \colon]_{\top_{ctx}} \sqsubseteq f \end{aligned}$$

CO-TOP

$$\frac{}{\mathit{ctx} \sqsubseteq \top_{\mathit{ctx}}}$$

CO-BOTTOM

$$\frac{}{\bot_{\mathit{ctx}} \sqsubseteq \mathit{ctx}}$$

CO-PARAMS

$$\frac{\begin{array}{c} \mathit{ctx}_1 = \,<(\mathit{bs}_1)\ \ldots> \\ \mathit{ctx}_2 = \,<(\mathit{bs}_2)\ \ldots> \qquad \mathit{bs}_1 \sqsubseteq \mathit{bs}_2 \qquad \mathit{ctx}_1[\texttt{self}] \sqsubseteq \mathit{ctx}_2.[\texttt{self}] \qquad \forall \mathit{var} : \mathit{ctx}_1[\mathit{var}] \sqsubseteq \mathit{ctx}_2[\mathit{var}] \end{array}}{\mathit{ctx}_1 \sqsubseteq \mathit{ctx}_2}$$

Figure 6.7: Comparison of Contexts

CJ-SYM

$$\frac{\mathit{ctx}_2 \sqcup \mathit{ctx}_1 = \mathit{ctx}_3}{\mathit{ctx}_1 \sqcup \mathit{ctx}_2 = \mathit{ctx}_3}$$

CJ-TOP

$$\frac{}{\mathit{ctx} \sqcup \top_{\mathit{ctx}} = \top_{\mathit{ctx}}}$$

CJ-BOTTOM

$$\frac{}{\mathit{ctx} \sqcup \bot_{\mathit{ctx}} = \mathit{ctx}}$$

CJ-DIFF

$$\frac{\mathit{ctx}_1 = \,<(\mathit{bs}_1)\ \ldots> \qquad \mathit{ctx}_2 = \,<(\mathit{bs}_2)\ \ldots> \qquad \mathit{bs}_1 \sqcup \mathit{bs}_2 = \top_{\mathit{bs}}}{\mathit{ctx}_1 \sqcup \mathit{ctx}_2 = \top_{\mathit{ctx}}}$$

CJ-PARAMS

$$\frac{\begin{array}{c} \mathit{ctx}_1 = \,<(\mathit{bs}_1)\ \ldots> \qquad \mathit{ctx}_2 = \,<(\mathit{bs}_2)\ \ldots> \qquad \mathit{ctx} = \,<(\mathit{bs})\ \ldots> \\ \mathit{bs} = \mathit{bs}_1 \sqcup \mathit{bs}_2 \qquad \mathit{bs} \neq \top_{\mathit{bs}} \\ \mathit{ctx}_1[\texttt{self}] \sqcup \mathit{ctx}_2[\texttt{self}] = \mathit{ctx}[\texttt{self}] \qquad \forall \mathit{var} :\ \mathit{ctx}_1[\mathit{var}] \sqcup \mathit{ctx}_2[\mathit{var}] = \mathit{ctx}[\mathit{var}] \end{array}}{\mathit{ctx}_1 \sqcup \mathit{ctx}_2 = \mathit{ctx}}$$

Figure 6.8: Join for Contexts

CM-SYM

$$\frac{\mathit{ctx}_2 \sqcap \mathit{ctx}_1 = \mathit{ctx}_3}{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \mathit{ctx}_3}$$

CM-TOP

$$\frac{}{\mathit{ctx} \sqcap \top_{\mathit{ctx}} = \mathit{ctx}}$$

CM-BOTTOM

$$\frac{}{\mathit{ctx} \sqcap \bot_{\mathit{ctx}} = \bot_{\mathit{ctx}}}$$

CM-DIFF

$$\frac{\begin{array}{c} \mathit{ctx}_1 = \,<(\mathit{bs}_1)\ \ldots> \qquad \mathit{ctx}_2 = \,<(\mathit{bs}_2)\ \ldots> \\ \mathit{bs}_1 \sqcap \mathit{bs}_2 = \bot_{\mathit{bs}} \end{array}}{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \bot_{\mathit{ctx}}}$$

CM-PARAMS

$$\frac{\begin{array}{c} \mathit{ctx}_1 = \,<(\mathit{bs}_1)\ \ldots> \qquad \mathit{ctx}_2 = \,<(\mathit{bs}_2)\ \ldots> \qquad \mathit{ctx} = \,<(\mathit{bs})\ \ldots> \\ \mathit{bs}_1 \sqcap \mathit{bs}_2 = \mathit{bs} \qquad \mathit{bs} \neq \bot_{\mathit{bs}} \\ \mathit{ctx}_1[\texttt{self}] \sqcap \mathit{ctx}_2[\texttt{self}] = \mathit{ctx}[\texttt{self}] \qquad \forall \mathit{var} :\ \mathit{ctx}_1[\mathit{var}] \sqcap \mathit{ctx}_2[\mathit{var}] = \mathit{ctx}[\mathit{var}] \end{array}}{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \mathit{ctx}}$$

Figure 6.9: Meet for Contexts

3. For all $act \in$ **all_activations**(cfg), and for all temporary variables var that are defined by act's block,

$$\begin{aligned} &cnt[act.temps\_cid][var.label] = object \\ &\quad \Rightarrow \; \exists ctx : \; ctx(act, cfg) \; \wedge \; [: V \; var :]_{ctx} \sqsubseteq f \end{aligned}$$

4. For all $act \in$ **all_activations**(cfg), and for all parameters var that are defined by act's block,

$$\begin{aligned} &cnt[act.params\_cid][var.label] = object \\ &\quad \Rightarrow \; \exists ctx : \; ctx(act, cfg) \; \wedge \; [: V \; var :]_{ctx} \sqsubseteq f \end{aligned}$$

5. For all $act \in$ **all_activations**(cfg),

$$\begin{aligned} &act.receiver = object \\ &\quad \Rightarrow \; \exists ctx : \; ctx(act, cfg) \; \wedge \; [: S \; act.block.method :]_{ctx} \sqsubseteq f \end{aligned}$$

For brevity, **flowpos**(*object*, *cfg*) refers to the least flow position that includes *object* in *cfg*. The calculation of **flowpos** for a particular *object* and *cfg* is straightforward: follow the above definition for an object being in a flow position, and create a union of precisely the required flow positions and no more.

6.6 Decomposition into simple data-flow structure

Simple data-flow structures are the subset of the data flow structures that do not use set elements. That is, a *simple type* is a class type, a block type whose context is a simple context, a selector type, $\top$, or $\bot$. A simple context is a parameters context whose types are all simple, $\top_{ctx}$, or $\bot_{ctx}$. A simple flow position is a method type whose context is simple, a variable type whose context is simple, or $\top_{fp}$ or $\bot_{fp}$.

Any data-flow structure may be decomposed into a number of simple elements using the **cpasplit** function, defined in Figure 6.13. Several properties are proven in section 6.11 to show that **cpasplit** does behave like a decomposition.

Decomposition is useful in **DDP** itself to subdivide the analysis of a general data-flow goal into a number of smaller goals. If future information requires that the original goal addresses a more general question, then the subgoals that have already been created are still useful, thus avoiding potentially wasted work. This form and this application of decomposition both follow Agesen, who describes the idea as follows [3]:

> The cartesian product algorithm (CPA for short) differs fundamentally. It does not partition sends, but instead turns the analysis of each send into a case analysis. To analyze a send, CPA computes the cartesian product of the types of the actual arguments. Each tuple in the cartesian product is analyzed as an independent case. This case analysis makes exact type information immediately available for each case, thus eliminating the need for iteration. In turn, the type information is used to ensure both precision (by avoiding type merges) and efficiency (by sharing cases to avoid redundant analysis). [. . .]
>
> The idea behind CPA is best understood by going back to the analogy between program execution and program analysis. During program execution, activation records are always created ”monomorphically,” simply because each slot contains a single object. Consider, for example, a polymorphic send expression that invokes the max: method with integer or float receivers. This means that sometimes the send invokes max: with an integer receiver, and other times it invokes it with a float receiver. But in any particular invocation the receiver is either an integer or a float: it cannot be both. We summarize this observation as follows:
>
> *There is no such thing as a polymorphic message, only polymorphic send expressions.*

$$\text{FO-TOP}\quad \frac{}{f \sqsubseteq \top_{fp}} \qquad \text{FO-BOTTOM}\quad \frac{}{\bot_{fp} \sqsubseteq f} \qquad \text{FO-VAR}\quad \frac{ctx \sqsubseteq ctx'}{[:\ V\ var\ :]_{ctx} \sqsubseteq [:\ V\ var\ :]_{ctx'}} \qquad \text{FO-METH}\quad \frac{ctx \sqsubseteq ctx'}{[:\ S\ \ meth\ :]_{ctx} \sqsubseteq [:\ S\ \ meth\ :]_{ctx'}}$$

$$\text{FO-SUM-L}\quad \frac{\forall f \in fs :\ f \sqsubseteq f'}{[:\ \Sigma\, fs\ :] \sqsubseteq f'} \qquad \text{FO-SUM-R}\quad \frac{\exists f' \in fs' :\ f \sqsubseteq f'}{f \sqsubseteq [:\ \Sigma\, fs'\ :]}$$

Figure 6.10: Comparison of Flow Positions

$$\frac{f_1 \sqsubseteq f_2}{f_1 \sqcup f_2 = f_2} \qquad \frac{f_2 \sqsubseteq f_1}{f_1 \sqcup f_2 = f_1} \qquad \frac{\begin{array}{c} f_1 \not\sqsubseteq f_2 \qquad f_2 \not\sqsubseteq f_1 \\ f_1 \textit{ is a variable or self flow position} \\ f_2 \textit{ is a variable or self flow position} \end{array}}{f_1 \sqcup f_2 = [:\ \Sigma\ \{f_1, f_2\}\ :]}$$

$$\frac{\begin{array}{c} f_1 \not\sqsubseteq f_2 \qquad f_2 \not\sqsubseteq f_1 \\ f_1 \textit{ is a variable or self flow position} \\ f_2 = [:\ \Sigma\, fs_2\ :] \\ fs = \textbf{remove\_redundecies}(\{f_1\} \cup fs_2) \end{array}}{f_1 \sqcup f_2 = [:\ \Sigma\, fs\ :]} \qquad \frac{\begin{array}{c} f_1 \not\sqsubseteq f_2 \qquad f_2 \not\sqsubseteq f_1 \\ f_2 \textit{ is a variable or self flow position} \\ f_1 = [:\ \Sigma\, fs_1\ :] \\ fs = \textbf{remove\_redundecies}(\{f_2\} \cup fs_1) \end{array}}{f_1 \sqcup f_2 = [:\ \Sigma\, fs\ :]}$$

$$\frac{\begin{array}{c} f_1 = [:\ \Sigma\, fs_1\ :] \qquad f_2 = [:\ \Sigma\, fs_2\ :] \\ fs = \textbf{remove\_redundecies}(fs_1 \cup fs_2) \end{array}}{f_1 \sqcup f_2 = [:\ \Sigma\, fs\ :]}$$

Figure 6.11: Join for Flow Positions

In addition to providing the groundwork for CPA context sensitivity, decomposition into simple data-flow objects is useful in the supporting theory.

6.7 Judgements

A data-flow algorithm processes *judgements* about the behavior of a program. This section describes the kinds of judgements that **DDP** processes.

6.7.1 Type judgements

A *type judgement* has the form $var :_{ctx} type$. If *ctx* specifies a block, that block must enclose the declaration of *var*. For example, if *var* is a global variable, then *ctx* may not specify a block and must be $\top_{ctx}$. Similarly, a context may only assign a type to the receiver if the variable is a parameter or a local variable; intuitively, there would otherwise there be no single `self` in scope.

The correctness criterion for type judgements is straightforward. A type judgement $var :_{ctx} type$ is correct for configuration *cfg* if:

$$\begin{aligned}
&\forall\, act \in \mathbf{all\_activations}(cfg) : \\
&\quad ctx(act, cfg) \;\wedge\; var = \mathbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, var.label) \\
&\qquad \Rightarrow \mathbf{read\_var}(cfg, act, var.label) \in type
\end{aligned}$$

That is, for every activation matched by *ctx* and in which *var* may be read at all, reading the specified variable gives an object included in *type*.

6.7.2 Simple flow judgements

A *simple flow judgement* $f \rightarrow f'$ declares that objects in flow position f may only directly flow to flow position f'. By definition, f may be any kind of flow position, but in **DDP** actually only processes flow judgements where f is a simple flow position.

No rigorous meaning is given to the correctness of an individual flow judgement, but the intuition is that $f \rightarrow f'$ means that f' holds all of the possible positions to which a value can directly flow if it starts in position f. The rigorous definition of correctness compares the flow position of an object to its flow position after one step of execution. Since a simple flow position cannot, in general, capture the entire flow position of an object at one configuration, **DDP** uses sets of flow judgements to capture all of the possible flow from one configuration to the next.

A *set* of simple flow judgements $\mathcal{F}$ is correct for configuration *cfg* precisely when:

$$\begin{aligned}
&\forall object \neq \texttt{NilObj} \,:\, \forall \mathcal{G} \subseteq \mathcal{F} : \\
&\quad \bot_{fp} \sqsubset \mathbf{flowpos}(object, cfg) \sqsubseteq \mathbf{lhs}(\mathcal{G}) \\
&\qquad \Rightarrow \quad \mathbf{flowpos}(object, \mathbf{step}_{\mathcal{P}}(cfg)) \;\sqsubseteq\; (\mathbf{lhs}(\mathcal{G}) \sqcup \mathbf{rhs}(\mathcal{G}))
\end{aligned}$$

where:

$$\mathbf{lhs}(\mathcal{G}) = \bigsqcup_{f \rightarrow f' \in \mathcal{G}} f$$

and:

$$\mathbf{rhs}(\mathcal{G}) = \bigsqcup_{f \rightarrow f' \in \mathcal{G}} f'$$

A set of flow judgements is correct for program $\mathcal{P}$, without the qualification, if it is correct in $\mathbf{step}_n(\mathcal{P})$ for all n.

6.7.3 Transitive flow judgements

A *transitive flow judgement* $f \rightarrow^* f'$ makes a stronger claim than a simple flow judgement: it claims that objects in flow position f may only flow to flow position f', even across an arbitrary number of **step**'s. A simple flow judgement makes a claim about one step of execution, while a transitive flow judgement makes a claim about an arbitrary number of steps of execution.

Formally, a set of transitive flow judgements $\mathcal{F}$ is *correct* for configurations cfg, $\textbf{step}_{\mathcal{P}}(cfg), \ldots, \textbf{step}^n_{\mathcal{P}}(cfg)$ whenever:

$$\begin{aligned}
&\forall object \neq \texttt{NilObj} : \ \forall i \in 0 \ldots n : \ \forall j \in i \ldots n : \ \forall \mathcal{G} \subseteq \mathcal{F} : \\
&\quad \bot_{fp} \sqsubset \textbf{flowpos}(object, \textbf{step}^i_{\mathcal{P}}(cfg)) \sqsubseteq \textbf{lhs}(\mathcal{G}) \\
&\quad\quad \Rightarrow \quad \textbf{flowpos}(object, \textbf{step}^j_{\mathcal{P}}(cfg)) \sqsubseteq \textbf{rhs}(\mathcal{G})
\end{aligned}$$

A set of transitive flow judgements is correct for program $\mathcal{P}$ if it is correct for the configurations $\textbf{step}_0(\mathcal{P})$, $\ldots$, $\textbf{step}_n(\mathcal{P})$ regardless of n.

6.7.4 Responders judgements

A *responders judgement* is one kind of judgement about the call graph. Roughly, it asks “what is invoked by a particular send statement?”. It has the following form:

$$stat_{ctx} \star b \xrightarrow{send} rs$$

The *stat* parameter must be a `send`, `sendvar`, or `beval` statement. b is a binding map for that statement, and *ctx* is a dynamic context. The parameter *rs* is typically a finite set of tuples $(bs, bctx)$, each of which has a block specification and a context. *rs* may also be $\top_r$, which signifies that any method or block might be invoked.

The above judgement is correct for configuration $\textbf{step}_n(\mathcal{P})$ if one of the following is true:

1. *rs* is $\top_r$.
2. *stat* is not the statement about to execute in $\textbf{step}_n(\mathcal{P})$.
3. b[`block`] is not the block of the main activation of $\textbf{step}_n(\mathcal{P})$.
4. *ctx* does not match the main activation of $\textbf{step}_n(\mathcal{P})$.
5. There is a tuple $(bs, bctx) \in rs$ such that the main activation in $\textbf{step}_{n+1}(\mathcal{P})$ has a block of *bs* and matches context *bctx*.

6.7.5 Senders judgements

A *senders judgement* is a different form of judgement about the call graph. It asks, roughly, “what statements invoke this block”. It has the following form:

$$bs_{ctx} \xleftarrow{send} ss$$

bs is the specification of a block or method and *ctx* is a context that filters execution states for the responding block. The presence of *ctx* thus allows more specific judgements about the calling context; it allows the analyzer to limit attention to the invokers of some block under the assumption that the execution state resulting from the block invocation matches the specified context.

The *senders set* in a senders judgement, *ss* in the example above, is typically a set of tuples $(stat, b, cctx)$, each of which specifies a statement with bindings and a context, but *ss* may also be the distinguished value $\top_s$. If $ss = \top_s$, the judgement declares that any statement might invoke *bs*.

The above judgement is correct for configuration $\textbf{step}_n(\mathcal{P})$ if one of the following is true:

1. ss is $\top_s$.

2. The next statement to execute in $\mathbf{step}_n(\mathcal{P})$ was not a `send`, `sendvar`, or `beval` statement.

3. The main activation of $\mathbf{step}_{n+1}(\mathcal{P})$ is for a block other than bs. In this case the senders judgement makes no claim.

4. There is a tuple $(stat, b, cctx)$ such that $cctx$ matches the main activation of $\mathbf{step}_n(\mathcal{P})$, the binding map of that activation is b, and the statement about to execute in that activation is $stat$. In this case, the sender is among the possibilities the judgement allows.

Lemma 6.2 (Senders Judgements Across Multiple Steps)**.** Suppose that $bs_{ctx} \xleftarrow{send} ss$ is correct for configurations $\mathbf{step}_0(\mathcal{P}) \dots \mathbf{step}_n(\mathcal{P})$. Then, for any activation $act \in \mathbf{all\_activations}(\mathbf{step}_{n+1}(\mathcal{P}))$ whose block is bs, that is matched by ctx, and whose $caller$ is not `undef`, there must be an $m \leq n$ such that $\mathbf{step}_m(\mathcal{P})$ matches the last criterion of correctness for senders judgements. That is, there is a tuple $(stat, b, cctx)$ such that $cctx$ matches the main activation of $\mathbf{step}_m(\mathcal{P})$, the binding map of that activation is b, and the statement about to execute in that activation is $stat$.

Proof. Disregarding changes to pc's, there is at most one new activation in each configuration that was not present in the previous configuration. Further, if there is a new activation at all, then the new activation must be the main activation. Therefore, every $act \in \mathbf{all\_activations}(\mathbf{step}_{n+1}(\mathcal{P}))$ must be the main activation of $\mathbf{step}_k(\mathcal{P})$ for some $k \leq n+1$. Further, since by assumption $caller \neq$ `undef`, act cannot be the initial activation created for $\mathbf{step}_0(\mathcal{P})$ and thus $k > 0$. Choose $m = k - 1$. Since this $m \leq n$, the senders judgement is true for $\mathbf{step}_m(\mathcal{P})$, and thus the final clause of the correctness criteria for senders judgement gives the desired property. □

6.8 Goals

There is one kind of *goal* for each kind of judgement described above. Each goal is a judgement that has had one portion removed:

- A type goal $v :_c ?$ tries to find a type t such that $v :_c t$ is correct.
- A flow goal $f \rightarrow ?$ tries to find a flow position f' such that $f \rightarrow f'$ is correct.
- A transitive flow goal $f \rightarrow^* ?$ tries to find a flow position f' such that $f \rightarrow^* f'$ is correct.
- A responders goal $stat_{ctx} \star b \xrightarrow{send} ?$ tries to find a responders set rs such that $stat_{ctx} \star b \xrightarrow{send} rs$ is correct.
- A senders goal $b_{ctx} \xleftarrow{send} ?$ tries to find a senders set ss such that $b_{ctx} \xleftarrow{send} ss$ is correct.

Every goal that **DDP** pursues is of one of the above five forms.

6.9 Restrictions

Not all elements of the above domains (types, contexts, etc.) are valid for use by **DDP**. There are some restrictions, both to ensure that valid elements have a meaningful interpretation, and to keep all of the domains finite. Some of the restrictions depend on the particular program $\mathcal{P}$ being analyzed.

Contexts have the bulk of the restrictions. To be consistent with the program, a context must only specify non-$\top$ types for parameters that are visible inside the context's block:

$$ctx[var] \neq \top \quad \Rightarrow \quad var \in \mathbf{static\_bindings}_{\mathcal{P}}(ctx.bs)$$

Further, not all contexts are usable for all purposes; there are additional restrictions as follow:

- For a type judgement $var :_{ctx} type$, either ctx must be $\top_{ctx}$, or it must specify a block surrounding the one where var is declared. For a global variable or instance variable, there is no such block and thus the context must be $\top_{ctx}$. For a block parameter or local variable, the context may specify the block where the variable is declared, or it may specify a block enclosing that block.
- A variable flow position $[\!: V\ var :\!]_{ctx}$ has exactly the same restrictions.
- For a self flow position $[\!: S\ meth :\!]_{ctx}$, either ctx must be $\top_{ctx}$, or it must specify the main block of $meth$.
- For a block type $B[\![blk]\!]_{ctx}$, either ctx must be $\top_{ctx}$, or it must specify a block that is blk itself or a block that surrounds blk.
- The context for a responders judgement is either $\top_{ctx}$, or it specifies a block that is the block of the binding map of the judgement or that lexically encloses that block.
- The context in a senders judgement is either $\top_{ctx}$, or it specifies a block that is the block of the judgement or that lexically encloses that block.

If a context is invalid according to either of the criteria above, then 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 that is valid for the intended purpose. In short, it is the smallest context ctx' such that ctx' is valid and $ctx \sqsubseteq ctx'$.

In detail, $\lceil ctx \rceil$ is computed as follows. First, if ctx is $\top_{ctx}$ or $\bot_{ctx}$, then $\lceil ctx \rceil = ctx$. Otherwise, the block specification of $\lceil ctx \rceil$ is the innermost block specification encompassing the block of ctx that meets the restrictions on block context; if there is no such block specification then $\lceil ctx \rceil = \top_{ctx}$. The non-$\top$ parameter restrictions of $\lceil ctx \rceil$ are precisely those of ctx where the variable is visible from the chosen block specification.

As another restriction, only classes, variables, methods, etc. in the program may be specified in the above domains. For example, in a type judgement $v :_c t$, the variable v must be a variable in $\mathcal{P}$.

Finally, the recursion between block types and contexts must be restricted in some way in order to ensure that a finite number of block types are possible. There are various approaches possible, as Agesen has studied [2]. The present work uses a simple approach, because it is expected that precise analysis of blocks that access themselves via parameters is not frequently needed for precise analysis of Smalltalk code. The approach used is as follows: the context associated with a block may not mention, either directly or indirectly, another block type for the same block. That is, while the context may mention a block type for a different block, the context for *that* block type may not mention a block type for either of the two blocks. And so on, recursively. Since there are only finitely many blocks available in a particular program, there are only a finite number of possible block types that meet this restriction. The notation $\lceil blkt \rceil$ refers to the block type $blkt$ after having enough contexts replaced by $\top_{ctx}$ that the block type is valid.

6.10 Proofs that the DDP domains are lattices

This section proves that block specifications, types, contexts, and flow positions all form lattices with their respective $\sqsubseteq$ operators, and that the respective $\sqcup$ and $\sqcap$ relations defined earlier are correct join and meet operations for these lattices. The theorem and proof are given first, so that the reader can see how the lemmas support it.

Theorem 2 (**DDP**'s data-flow domains are lattices)**.** *Block specifications, types, contexts, and flow positions, along with their respective $\sqsubseteq$ relations, are lattices. Further, the respective $\sqcup$ and $\sqcap$ operations defined in this chapter are these lattices' join and meet operations.*

Proof. The following lemmas establish that each $\sqsubseteq$ relation is a true partial order: each is reflexive, antisymetric, and transitive. Further, the following lemmas show that each $\sqcup$ and $\sqcap$ relation provides least upper bounds (LUB's) and greatest lower bounds (GLB's). Inspection of the definitions of these relations are complete. Thus, not only does each data-flow domain have LUB's and GLB's, but the specified $\sqcup$ and $\sqcap$ relations provide them. Since LUB's and GLB's are unique, the defined $\sqcup$ and $\sqcap$ relations are operations. □

Lemma 6.3 (Comparison of Block Specifications is Reflexive)**.** For any block specification *bs*, $bs \sqsubseteq bs$.

Proof. Straightforward case analysis of *bs* gives the desired property. □

Lemma 6.4 (Comparison of Block Specifications is Transitive)**.** For any block specifications bs_1, bs_1, and bs_1, where $bs_1 \sqsubseteq bs_2$ and $bs_2 \sqsubseteq bs_3$, $bs_1 \sqsubseteq bs_3$.

Proof. The only non-trivial case is when none of bs_1, bs_2, and bs_3 are either $\top_{bs}$ or $\bot_{bs}$. Let:

$$\begin{aligned} bs_1 &= (ms_1, l_1) \\ bs_2 &= (ms_2, l_2) \\ bs_3 &= (ms_3, l_3) \end{aligned}$$

Rule BSO-NESTED must have been used, so that all of:

$$\begin{aligned} ms_1 &= ms_2 \\ l_1 &= l_2@l' \\ ms_2 &= ms_3 \\ l_2 &= l_3@l'' \end{aligned}$$

for some l' and l''. It is thus clear that $ms_1 = ms_3$ and that $l_1 = l_3@(l'@l'')$, and thus $bs_1 \sqsubseteq bs_3$. □

Lemma 6.5 (Comparison of Block Specifications is Antisymmetric)**.** For any block specifications bs_1 and bs_2, if $bs_1 \sqsubseteq bs_2$ and $bs_2 \sqsubseteq bs_1$, then $bs_1 = bs_2$.

Proof. The proof is a case analysis of bs_1. If bs_1 is $\top_{bs}$, then BSO-TOP must have been used to justify $bs_1 \sqsubseteq bs_2$, and thus bs_2 is $\top_{bs}$. Likewise for the case where bs_1 is $\bot_{bs}$.

If $bs_1 = (ms, l)$, then BSO-NESTED must have been used to justify $bs_1 \sqsubseteq bs_2$ and $bs_2 \sqsubseteq bs_1$. The only way this can be is if $bs_2 = (ms, l@l')$ where $l' = []$. Thus $bs_1 = bs_2$. □

Lemma 6.6 (Join of Block Specifications is Complete)**.** For any block specifications bs_1 and bs_2, there is a $bs_\sqcup$ such that $bs_1 \sqcup bs_2 = bs_\sqcup$.

Proof. The proof is by cases. If bs_1 or bs_2 is $\top_{bs}$, then one may choose $bs_\sqcup = \top_{bs}$. Similarly, if $bs_1 = \bot_{bs}$ then one may choose $bs_\sqcup = bs_2$, and if $bs_2 = \bot_{bs}$ then one may choose $bs_\sqcup = bs_1$. That leaves the case that neither bs_1 nor bs_2 is $\top_{bs}$ or $\bot_{bs}$. In that case, one must consider whether the methods of the two block specifications are the same. If they are the same, then BSJ_SAMEMETH may be used to find a satisfactory $bs_\sqcup$, and if they are different then one may choose $\top_{bs}$. □

Lemma 6.7 (Meet of Block Specifications is Complete)**.** For any block specifications bs_1 and bs_2, there is a $bs_\sqcap$ such that $bs_1 \sqcap bs_2 = bs_\sqcup$.

Proof. The proof is by cases, just as with the proof that $\sqcup$ is complete. □

Lemma 6.8 (Join of Block Specifications is Correct)**.** If $bs_1 \sqcup bs_2 = bs_\sqcup$, then $bs_\sqcup$ is the least upper bound of bs_1 and bs_2. That is, $bs_1 \sqsubseteq bs_\sqcup$ and $bs_2 \sqsubseteq bs_\sqcup$, and for any other bs_3 for which $bs_1 \sqsubseteq bs_3$ and $bs_2 \sqsubseteq bs_3$, it must be that $bs_\sqcup \sqsubseteq bs_3$.

Proof. To show the first part of the lemma, that $bs_\sqcup$ is an upper bound of bs_1 and bs_2, induct on the derivation that $bs_1 \sqcup bs_2 = bs_3$ and note that, in each case, one of the ordering rules will clearly apply.

To show that $bs_\sqcup$ is also the *least* upper bound of bs_1 and bs_2, induct on the derivation that $bs_1 \sqcup bs_2 = bs_3$ and consider any other upper bound bs_3. It must be shown that $bs_\sqcup \sqsubseteq bs_3$. The only non-trivial case is if the derivation finishes with BSJ-SAMEMETH. In that case, let:

$$\begin{aligned} bs_1 &= (ms, l_1) \\ bs_2 &= (ms, l_2) \\ bs_\sqcup &= (ms, l_\sqcup) \\ l_\sqcup &= \textbf{longest\_prefix}(l_1, l_2) \end{aligned}$$

If $bs_3 = \top_{bs}$ then the result is trivial, and if $bs_3 = \bot_{bs}$ then there is a contradiction because bs_3 cannot be an upper bound of bs_1 or bs_2. Suppose, then, that:

$$bs_3 = (ms, l_3)$$

The justification that $bs_1 \sqsubseteq bs_3$ must use BSO-NESTED, and thus $l_1 = l_3@l'$ for some l'. Likewise, $l_2 = l_3@l''$ for some l''. Thus, l_3 is a prefix of both l_1 and l_2, and thus also a prefix of $l_\sqcup$, which is the longest common prefix of l_1 and l_2. Thus by BSO-NESTED, $bs_\sqcup \sqsubseteq bs_3$. □

Lemma 6.9 (Meet of Block Specifications is Correct)**.** When $bs_1 \sqcap bs_2 = bs_\sqcap$, $bs_\sqcap$ is the greatest lower bound of bs_1 and bs_2. That is, $bs_\sqcap \sqsubseteq bs_1$ and $bs_\sqcap \sqsubseteq bs_2$, and for any other bs_3 for which $bs_3 \sqsubseteq bs_1$ and $bs_3 \sqsubseteq bs_2$, it must be that $bs_3 \sqsubseteq bs_\sqcap$.

Proof. It is straightforward to show the first part of the lemma, that $bs_\sqcap \sqsubseteq bs_1$ and $bs_\sqcap \sqsubseteq bs_2$. Simply induct on the derivation that $bs_1 \sqcap bs_2 = bs_\sqcap$.

To show the second part, induct on the derivation that $bs_1 \sqcap bs_2 = bs_\sqcap$, and let bs_3 be such that $bs_3 \sqsubseteq bs_1$ and $bs_3 \sqsubseteq bs_2$. It must be shown that $bs_3 \sqsubseteq bs_\sqcap$. Consider each possible last step of the derivation that $bs_1 \sqcap bs_2 = bs_\sqcap$:

- BSM-SYM. By the inductive assumption, $bs_3 \sqsubseteq bs_\sqcap$.
- BSM-TOP. By BSO-TOP, $bs_3 \sqsubseteq bs_\sqcap = \top_{bs}$.
- BSM-BOTTOM. Since $bs_3 \sqsubseteq bs_2 = \bot_{bs}$, it must be that $bs_3 = \bot_{bs}$. By BSO-BOTTOM, $bs_3 \sqsubseteq bs_\sqcap$.
- BSM-DIFFMETH. Let $bs_1 = (ms_1, l_1)$ and $bs_1 = (ms_2, l_2)$. If $bs_3 = \bot_{bs}$ then the proof is trivial, and if $bs_3 = \top_{bs}$ then it cannot be that $bs_3 \sqsubseteq bs_1$. Thus suppose $bs_3 = (ms_3, l_3)$. To justify that $bs_3 \sqsubseteq bs_1$, rule BSO-NESTED must be used, and thus $ms_3 = ms_1$. The same argument applies with bs_2, however, and thus also $ms_3 = ms_2$. This contradicts the assumption of BSM-DIFFMETH, and thus the case is impossible.
- BSM-DIFFBLOCK. Again, the only non-trivial case is if $bs_3 = (ms_3, l_3)$ for some ms_3 and l_3. Let:

  $$\begin{aligned} bs_1 &= (ms, l@[a]@l_1) \\ bs_2 &= (ms, l@[b]@l_2) \\ a &\neq b \end{aligned}$$

  Rule BSO-NESTED must be used to justify both $bs_3 \sqsubseteq bs_1$ and $bs_3 \sqsubseteq bs_2$, and thus $ms_3 = ms$. Further, it must be that both:

  $$l_3 = l@[a]@l_1@l'$$
  $$l_3 = l@[b]@l_2@l'$$

However, this is clearly impossible.

- BSM-NESTED. Again, the only non-trivial case is if $bs_3 = (ms_3, l_3)$ for some ms_3 and l_3. Let:

$$\begin{array}{rcl} bs_1 & = & (ms, l) \\ bs_2 & = & (ms, l@l') \\ bs_\sqcap & = & (ms, l@l') \end{array}$$

Rule BSO-NESTED must be used to justify $bs_3 \sqsubseteq bs_1$ and $bs_3 \sqsubseteq bs_2$, and thus $ms_3 = ms$. Further, $l_3 = l@l'@l''$. Thus, by Rule BSM-NESTED, $bs_3 \sqsubseteq bs_\sqcap$.

□

Lemma 6.10 (Reflexive Property for Comparison of Types)**.** For any type t, $t \sqsubseteq t$.

Proof. Justification rule TO-REFL gives this property directly. □

Lemma 6.11 (Comparison of Contexts is Reflexive)**.** For any context ctx, it must be that $ctx \sqsubseteq ctx$.

Proof. If $ctx = \top_{ctx}$, then CO-TOP gives this property. Likewise, if $ctx = \bot_{ctx}$, then CO-BOTTOM gives the property. If ctx is a parameters context, then CO-PARAMS gives the desired property. □

Lemma 6.12 (Comparison of Types and Contexts is Transitive)**.** For any types t_1, t_2, and t_3, where $t_1 \sqsubseteq t_2$, and $t_2 \sqsubseteq t_3$, it must be that $t_1 \sqsubseteq t_3$. For any contexts ctx_1, ctx_2, and ctx_3, where $ctx_1 \sqsubseteq ctx_2$ and $ctx_2 \sqsubseteq ctx_3$, it must be that $ctx_1 \sqsubseteq ctx_3$.

Proof. The proof is by induction on the depth of the deepest justification that $t_1 \sqsubseteq t_2$, $t_2 \sqsubseteq t_3$, $ctx_1 \sqsubseteq ctx_2$, or $ctx_2 \sqsubseteq ctx_3$.

First, consider the transitivity of types. One of the following rules must be used to justify that $t_1 \sqsubseteq t_2$:

- TO-REFL. Then $t_1 = t_2$. Since, by assumption, $t_2 \sqsubseteq t_3$, it must also be that $t_1 \sqsubseteq t_3$.
- TO-TOP. Then $t_2 = \top$. It must also be that $t_3 = \top$, because otherwise it is not possible for $t_2 \sqsubseteq t_3$. By TO-TOP again, $t_1 \sqsubseteq t_3$.
- TO-BOTTOM. Then $t_1 = \bot$. By TO-BOTTOM again, $t_1 \sqsubseteq t_3$ regardless of what t_3 is.
- TO-BLOCK-CTX. Then $t_1 = B\llbracket bs \rrbracket_{ctx_1}$ and $t_2 = B\llbracket bs \rrbracket_{ctx_2}$, where $ctx_1 \sqsubseteq ctx_2$. Now consider each way that it might have been justified that $t_2 \sqsubseteq t_3$:
  - TO-TOP. Then $t_3 = \top$, and by TO-TOP, $t_1 \sqsubseteq t_3$.
  - TO-BLOCK-CTX. Then $t_3 = B\llbracket bs \rrbracket_{ctx_3}$ where $ctx_2 \sqsubseteq ctx_3$. The derivations that $ctx_1 \sqsubseteq ctx_2$ and that $ctx_2 \sqsubseteq ctx_3$ must be less deep than the derivations that $t_1 \sqsubseteq t_2$ and that $t_2 \sqsubseteq t_3$. Thus the inductive assumption may be used, and $ctx_1 \sqsubseteq ctx_3$. Thus by TO-BLOCK-CTX, $t_1 \sqsubseteq t_3$.
  - TO-BLOCK-CLASS. Then TO-BLOCK-CLASS also justifies $t_1 \sqsubseteq t_3$.
  - TO-SUM-R. Then t_3 is a sum type that has a t' among its elements such that $t_2 \sqsubseteq t'$. By the inductive assumption, it is also true that $t_1 \sqsubseteq t'$. Thus by TO-SUM-R, $t_1 \sqsubseteq t_3$.
- TO-BLOCK-CLASS. t_1 is a block type and t_2 is the class type for class Block. Consider each way to justify $t_2 \sqsubseteq t_3$:
  - TO-REFL. Then t_3 is also the class type for Block and the desired result is given by TO-BLOCK-CLASS.
  - TO-TOP. Then $t_3 = \top$, and TO-TOP gives the desired result.
  - TO-SUM-R. Then the inductive assumption gives the desired result.

- TO-SELECTOR. The proof parallels the one for TO-BLOCK-CLASS.
- TO-SUM-R. t_2 is a sum type, and there is some element t' of the sum for which $t_1 \sqsubseteq t'$. Consider each way that it may have been justified that $t_2 \sqsubseteq t_3$:
  - TO-REFL. Trivial.
  - TO-TOP. Trivial.
  - TO-SUM-R. Then t_3 is a sum type with an element t'' such that $t_2 \sqsubseteq t''$. To justify that $t_2 \sqsubseteq t''$, one must use TO-SUM-L. Thus, all elements of t_2 are subtypes of t'', including t'. By the inductive assumption, $t_1 \sqsubseteq t''$, and thus by TO-SUM-R, $t_1 \sqsubseteq t_3$.
  - TO-SUM-L. All elements of t_2 are subtypes of t_3, including t'. By the inductive assumption, $t_1 \sqsubseteq t_3$.
- TO-SUM-L. t_1 is a sum type, and every element of the sum is a subtype of t_2. Each element of t_1 cannot be a sum type, and so the justification that each element is a subtype of t_2 cannot use TO-SUM-L and must instead use one of the above rules. No matter which rule is used, the argument from above may be repeated to show that the element is also a subtype of t_3. Thus the condition is met to use TO-SUM-L to justify $t_1 \sqsubseteq t_3$.

Now consider transitivity of contexts. One of the following rules must be used to justify that $ctx_1 \sqsubseteq ctx_2$:

- CO-TOP. Then $ctx_2 = \top_{ctx}$. It most also be that $ctx_3 = \top_{ctx}$, and thus clearly $ctx_1 \sqsubseteq ctx_3$.
- CO-BOTTOM. Then $ctx_1 = \bot_{ctx}$. Then by CO-BOTTOM, $ctx_1 \sqsubseteq ctx_3$.
- CO-PARAMS. Then ctx_1 and ctx_2 are parameters contexts where the types of ctx_1 are subtypes of the corresponding types of ctx_2. If one justifies $ctx_2 \sqsubseteq ctx_3$ with CO-TOP then the result is trivial, so suppose one uses CO-PARAMS. Then the types of ctx_2 are subtypes of the corresponding types of ctx_3. By the inductive assumption, the types of ctx_1 are also less than the corresponding types of ctx_3, and thus CO-PARAMS justifies that $ctx_1 \sqsubseteq ctx_3$.

□

Lemma 6.13 (Comparison of Types and Contexts is Antisymmetric)**.** Let t_1 and t_2 be any types, and ctx_1 and ctx_2 be any contexts. If $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_1$ then $t_1 = t_2$. If $ctx_1 \sqsubseteq ctx_2$ and $ctx_2 \sqsubseteq ctx_1$ then $ctx_1 = ctx_2$.

Proof. The proof is by induction on the depth of the deepest inference tree used to infer that $t_1 \sqsubseteq t_2$, $t_2 \sqsubseteq t_1$, $ctx_1 \sqsubseteq ctx_2$, or $ctx_2 \sqsubseteq ctx_1$.

First consider anti-symmetry of comparison of types. Consider in turn each way that one might justify $t_1 \sqsubseteq t_2$:

- TO-REFL. It must be that $t_1 = t_2$ in order to use this rule at all.
- TO-TOP. Thus $t_2 = \top$. The only ways to justify that $\top \sqsubseteq t_1$ are TO-REFL, TO-TOP, and TO-SUM-R. If TO-REFL or TO-TOP is used the result is trivial. TO-SUM-R cannot in fact be used because the assumptions cannot be met: There is no way for the sum to include a type that is a supertype of $\top$.
- TO-BOTTOM. Likewise.
- TO-BLOCK-CTX. It must be that $t_1 = B\{bs\}_{bctx_1}$ and $t_2 = B\{bs\}_{bctx_2}$ for some bs, $bctx_1$, and $bctx_2$. To justify that $t_2 \sqsubseteq t_1$, either TO-REFL is used, or TO-BLOCK-CTX is used again. If TO-REFL is used then the result is trivial. If TO-BLOCK-CTX is used, then it must be that both $bctx_1 \sqsubseteq bctx_2$ and $bctx_2 \sqsubseteq bctx_1$. By the inductive hypothesis, $bctx_1 = bctx_2$ and thus also $t_1 = t_2$.
- TO-BLOCK-CLASS. In this case, there is no rule that can justify $t_2 \sqsubseteq t_1$.

- TO-SELECTOR. Likewise.
- TO-SUM-R. It must be that $t_2 = \Sigma ts_2$ and that there is a $t'_2 \in ts_2$ such that $t_1 \sqsubseteq t'_2$. To justify $t_2 \sqsubseteq t_1$, one of these rules must be used: TO-REFL, TO-TOP, TO-SUM-R, or TO-SUM-L. TO-REFL is trivial, and, as described above, TO-TOP is impossible. In fact, as described below, TO-SUM-L and TO-SUM-R are impossible as well.

  If TO-SUM-R is used, then $t_1 = \Sigma ts_1$ and there is a $t'_1 \in ts_1$ such that $t_2 \sqsubseteq t'_1$. The only way to justify $t_2 \sqsubseteq t'_1$ is with TO-SUM-L, which means that every element of ts_2, including t'_2, is a subtype of t'_1. By the inductive hypothesis, $t'_1 = t'_2$, and thus every element of ts_2 is also a subtype of t'_2. But then t_2 is a malformed sum type.

  If TO-SUM-L is used, then every element of ts_2 is a subtype of t_1. By the transitivity property, every element of ts_2 must also be a subtype of t'_2. But then t_2 is, again, a malformed sum type.
- TO-SUM-L. It must be that $t_1 = \Sigma ts_1$ and that all elements of ts_1 are subtypes of t_2. To justify that $t_2 \sqsubseteq t_1$, one of these rules must be used: TO-REFL, TO-BOTTOM, TO-SUM-R, or TO-SUM-L. TO-REFL is trivial. TO-BOTTOM is actually impossible because no sum type can be a subtype of $\bot$. TO-SUM-R is symmetric to a case already discussed.

  That leaves TO-SUM-L to justify $t_2 \sqsubseteq t_1$. It must be that $t_2 = \Sigma ts_2$ and every element of ts_2 is a subtype of t_1. To justify that each of these elements is a subtype of t_1, TO-SUM-R must be used. Thus for each element of ts_2, there must be an element of ts_1 that it is a subtype of. Similarly, there must be an element of ts_1 that is the supertype of each element of ts_2. Consider any element $t_{1a} \in ts_1$, an element $t_{2a} \in ts_2$ such that $t_{1a} \sqsubseteq t_{2a}$, and an element $t_{1b} \in ts_1$ such that $t_{2a} \sqsubseteq t_{1b}$. By transitivity, $t_{1a} \sqsubseteq t_{1b}$. Since t_1 is a well-formed sum type, it must be that $t_{1a} = t_{1b}$. By the inductive hypothesis, it must be that $t_{1a} = t_{2a}$. Since this argument holds for all elements of ts_1 and ts_2, it must be that each element of each set has an equal element in the other set, and thus the two sets must be equal. Thus $t_1 = t_2$ as well.

Now consider comparision of contexts. Consider in turn each rule that can justify $ctx_1 \sqsubseteq ctx_2$:

- CO-TOP. Then $ctx_2 = \top_{ctx}$. Since $ctx_2 \sqsubseteq ctx_1$, it must also be that $ctx_2 = \top_{ctx}$; no other type can be $\sqsubseteq \top_{ctx}$. Thus $ctx_1 = ctx_2$.
- CO-BOTTOM. Likewise, $ctx_1 = \bot_{ctx}$, and $ctx_2 = \bot_{ctx}$ as well.
- CO-PARAMS. In this case, ctx_1 and ctx_2 must both be parameters contexts. Thus, CO-PARAMS must also have been used to justify $ctx_2 \sqsubseteq ctx_1$. Since block-specification comparison is anti-symmetric, this implies that the block specification of ctx_1 is the same as that of ctx_2. Likewise, since type comparison is anti-symmetric, all of the types in ctx_1 must equal the corresponding types in ctx_2. Therefore, all of the components of ctx_1 equal the corresponding components of ctx_2, and $ctx_1 = ctx_2$.

□

Lemma 6.14 (Join of Types is Complete). For any types t_1 and t_2, there is a type $t_\sqcup$ such that $t_1 \sqcup t_2 = t_\sqcup$.

Proof. The proof is straightforward by cases. Either one type is a subtype of the other, or if not, each type is or is not a sum type. □

Lemma 6.15 (Join of Contexts is Complete). For any contexts ctx_1 and ctx_2, there is a context $ctx_\sqcup$ such that $ctx_1 \sqcup ctx_2 = ctx_\sqcup$.

Proof. The proof by cases is straightforward. □

Lemma 6.16 (Meet of Types and Contexts is Complete). For any types t_1 and t_2, there is a type $t_\sqcup$ such that $t_1 \sqcap t_2 = t_\sqcup$. For any contexts ctx_1 and ctx_2, there is a context $ctx_\sqcup$ such that $ctx_1 \sqcap ctx_2 = ctx_\sqcup$.

Proof. The proof is by induction on the construction of t_1 and t_2 or ctx_1 and ctx_2. Each type is $\top$, $\bot$, a class type, a selector type, a block type, or a sum type. In each case, either one type is a subtype of the other, or one of the rules must apply. Note that if t_1 and t_2 are blocks, then it will be possible to satisfy the assumption in TM-BLOCK-SAME due to the inductive assumption. Likewise for TM-SUM. Similarly, each context is $\top_{ctx}$, $\bot_{ctx}$, or a parameters context. Likewise, note that the assumptions in CM-PARAMS are satisfiable due to the inductive assumption. □

Lemma 6.17 (Join of Types is Correct)**.** If $t_1 \sqcup t_2 = t_\sqcup$, $t_\sqcup$ is the least upper bound of t_1 and t_2 in the types lattice. That is, $t_1 \sqsubseteq t_\sqcup$, $t_2 \sqsubseteq t_\sqcup$, and for any $t'_\sqcup$ such that $t_1 \sqsubseteq t'_\sqcup$ and $t_2 \sqsubseteq t'_\sqcup$, $t_\sqcup \sqsubseteq t'_\sqcup$.

Proof. To show that $t_1 \sqsubseteq t_\sqcup$ and $t_2 \sqsubseteq t_\sqcup$, one can do a straightforward case analysis on the derivation of $t_1 \sqcup t_2 = t_\sqcup$. Thus $\sqcup$ gives upper bounds.

To show that $t_\sqcup$ is the *least* upper bound, let $t'_\sqcup$ be any other upper bound, and consider each way that it may be derived that $t_1 \sqcup t_2 = t_\sqcup$.

- TJ-SUB1 and TJ-SUB2. Trivial.

- TJ-SIMPLES. By TO-SUM-L, it must be that $t_\sqcup \sqsubseteq t'_\sqcup$; the two elements of $t_\sqcup$ are t_1 and t_2 which are assumed to be subtypes of $t'_\sqcup$.

- TJ-MIXED1. t_2 and $t_\sqcup$ must be sum types, and every component type of $t_\sqcup$ must also be a component of t_2 or must be exactly t_1. Each of these types must be a subtype of $t'_\sqcup$, and thus again TO-SUM-L shows that $t_\sqcup \sqsubseteq t'_\sqcup$.

- TJ-MIXED2. Likewise.

- TJ-SUMS. t_1, t_2, and $t_\sqcup$ are all sum types, and every component of $t_\sqcup$ is a component of either t_1 or t_2. Thus TO-SUM-L again shows that $t_\sqcup \sqsubseteq t'_\sqcup$.

Lemma 6.18 (Join of Contexts is Correct)**.** For any three contexts ctx_1, ctx_2, and $ctx_\sqcup$, where $ctx_1 \sqcup ctx_2 = ctx_\sqcup$, $ctx_\sqcup$ is the least upper bound of ctx_1 and ctx_2.

First show that $ctx_\sqcup$ is an upper bound, induct on the derivation of $ctx_1 \sqcup ctx_2 = ctx_\sqcup$, and observe that in each possible case, one of the ordering rules will apply.

To show that $ctx_\sqcup$ is not only an upper bound, but the least upper bound, induct again on the derivation of $ctx_1 \sqcup ctx_2 = ctx_\sqcup$, consider each possible rule that might be used in the final step of the derivation, and consider any other upper bound $ctx'_\sqcup$. It must be shown that $ctx_\sqcup \sqsubseteq ctx'_\sqcup$.

- CJ-SYM. The inductive assumption directly gives the desired result.

- CJ-TOP. Every context is ordered before $\top_{ctx}$.

- CJ-BOTTOM. $ctx'_\sqcup$ can only be $\bot_{ctx}$; otherwise there is no way to justify ordering it before $\bot_{ctx}$. Thus by CO-BOTTOM, $ctx_\sqcup \sqsubseteq ctx'_\sqcup$.

- CJ-DIFF. If $ctx'_\sqcup$ is $\top_{ctx}$, then the proof is trivial. It cannot be $\bot_{ctx}$. If $ctx'_\sqcup$ is a parameters context, then a contradiction arises, as follows. The block specification of $ctx'_\sqcup$ must be ordered after both the block specification of ctx_1 and the block specification of ctx_2. Since the join operation for block specifications gives least upper bounds, the only context this could be is $\top_{ctx}$. However, the block specification of a parameters context can not be $\top_{ctx}$.

- CJ-PARAMS. If $ctx'_\sqcap$ is $\top_{ctx}$, then the proof is trivial. Further, $ctx'_\sqcap$ cannot be $\bot_{ctx}$. The only remaining case is that $ctx'_\sqcap = <\ (bs_3)\ \ldots >$. It must be that all of:

$$\begin{aligned} bs_1 &\sqsubseteq bs_3 \\ bs_2 &\sqsubseteq bs_3 \\ ctx_1[\mathtt{self}] &\sqsubseteq ctx_\sqcup[\mathtt{self}] \\ ctx_2[\mathtt{self}] &\sqsubseteq ctx_\sqcup[\mathtt{self}] \\ \forall var:\ ctx_1[var] &\sqsubseteq ctx'_\sqcup[var] \\ \forall var:\ ctx_2[var] &\sqsubseteq ctx'_\sqcup[var] \end{aligned}$$

Since meet is correct for block specifications, $bs_1 \sqcup bs_2 \sqsubseteq bs_3$. By the inductive assumption, both:

$$\begin{aligned} ctx_1[\mathtt{self}] \sqcup ctx_2[\mathtt{self}] &\sqsubseteq ctx'_\sqcup[\mathtt{self}] \\ \forall var:\ ctx_1[var] \sqcup ctx_2[var] &\sqsubseteq ctx'_\sqcup[var] \end{aligned}$$

Thus all of the conditions are met to use rule CO-PARAMS, and $ctx_\sqcup \sqsubseteq ctx'_\sqcup$.

□

Lemma 6.19 (Meet of Types and Contexts is Correct)**.** For any types t_1, t_2, and $t_\sqcap$, where $t_1 \sqcap t_2 = t_\sqcap$, $t_\sqcap$ is the greatest lower bound of t_1 and t_2 in the types lattice. That is, $t_\sqcap \sqsubseteq t_1$, $t_\sqcap \sqsubseteq t_2$, and for any $t'_\sqcap$ such that $t'_\sqcap \sqsubseteq t_1$ and $t'_\sqcap \sqsubseteq t_2$, $t'_\sqcap \sqsubseteq t_\sqcap$. Further, for any contexts ctx_1, ctx_2, and $ctx_\sqcap$ where $ctx_1 \sqcap ctx_2 = ctx_\sqcap$, $ctx_\sqcap$ is the least upper bound of ctx_1 and ctx_2.

Proof. First show that $t_\sqcap$ and $ctx_\sqcap$ are lower bounds. It suffices to perform a straightforward induction on the derivation of $t_1 \sqcap t_2 = t_\sqcap$ or $ctx_1 \sqcap ctx_2 = ctx_\sqcap$. The only non-trivial case is TM-SUM. One can show, by a tedious case analysis, that whenever two types are a subtype of a third type, their union is also a subtype of the third type. By extension, if any finite number of types is a subtype of t, then so is their union.

Thus $\sqcap$ gives lower bounds.

To show that $t_\sqcap$ and $ctx_\sqcap$ are *greatest* lower bounds, let $t'_\sqcap$ and $ctx'_\sqcap$ be any other lower bounds. It must be shown that $t'_\sqcap \sqsubseteq t_\sqcap$ and that $ctx'_\sqcap \sqsubseteq ctx_\sqcap$.

First, consider each way that it may be derived that $t_1 \sqcap t_2 = t_\sqcap$.

- TM-SYM. The inductive assumption gives the desired result.
- TM-SUBTYPE. Since $t'_\sqcap$ is a lower bound, it must be that $t'_\sqcap \sqsubseteq t_1 = t_\sqcap$.
- TM-CLASS. A case analysis on the different kinds of types shows that $t'_\sqcap = \bot$. Since this argument is used repeatedly, this simple case analysis will be listed in full:
  - If $t'_\sqcap$ is $\top$, then there is a contradiction: there is no way to justify $t'_\sqcap \sqsubseteq t_1$.
  - If $t'_\sqcap$ is a class type, then it must be a class type for the same class as t_1 and for the same class as t_2. However, by an assumption of TM-CLASS, these classes are different.
  - If $t'_\sqcap$ is a selector type, then the class of t_1 and the class of t_2 must be Selector. However, these classes cannot be the same.
  - If $t'_\sqcap$ is a block type, then likewise for class Block.
  - If $t'_\sqcap$ is a sum type, then all of its elements must be subtypes of t_1 and t_2. However, its elements must be class, selector, or block types, and as argued above there are no such types available that are subtypes of both t_1 and t_2.

Since $t'_\sqcap = \bot = t_\sqcap$, $t'_\sqcap \sqsubseteq t_\sqcap$.

- TM-CLASS-SELECTOR. Likewise.
- TM-CLASS-BLOCK. Likewise.
- TM-SELECTOR1. Likewise.
- TM-SELECTOR2. Likewise.
- TM-SEL-BLOCK. Likewise.
- TM-BLOCK-DIFF. Likewise.
- TM-BLOCK-SAME. If $t'_\sqcap = \bot$, then the desired result is clear. Otherwise, $t'_\sqcap$ must be a block type. Its block specification must be the same as that for t_1 and t_2, and its context must be subsumed by both the context of t_1 and the context of t_2. By the inductive assumption, any such context must be subsumed by the meet of these two contexts, and the context of $t_\sqcap$ is in fact the meet of these two contexts. Thus, by TO-BLOCK, $t'_\sqcap \sqsubseteq t_\sqcap$.
- TM-SUM. $t_1 = \Sigma ts$. Since $t'_\sqcap \sqsubseteq t_1$, there must be some element $t' \in ts$ such that $t'_\sqcap \sqsubseteq t'$. By the inductive assumption, $t'_\sqcap \sqsubseteq t' \sqcap t_2$. Since $t_\sqcap$ is the join of $t' \sqcap t_2$ with some other types, and, since joins give upper bounds, it must also be that $t'_\sqcap \sqsubseteq t_\sqcap$.

Now consider each way to derive $ctx_1 \sqcap ctx_2 = ctx_\sqcap$.

- CM-SYM. By the inductive assumption, $ctx'_\sqcap \sqsubseteq ctx_\sqcap$.
- CM-TOP. Then $ctx_2 = \top_{ctx}$ and $ctx_\sqcap = ctx_1$. By assumption, $ctx'_\sqcap \sqsubseteq ctx_1 = ctx_\sqcap$.
- CM-BOTTOM. It must be that $ctx'_\sqcap = \bot_{ctx}$. Thus by CO-BOTTOM, $ctx'_\sqcap \sqsubseteq ctx_\sqcap$.
- CM-DIFF. If $ctx'_\sqcap$ is $\bot_{ctx}$, then the proof is trivial, and $ctx'_\sqcap$ cannot be $\top_{ctx}$. If it is a parameters context, a contradiction arises. Its block specification must be a subtype of the block specifications of ctx_1 and ctx_2, and its block specification must not be $\bot_{bs}$. By the inductive assumption, however, and since meet for block specifications gives the greatest lower bound, its block specification can only be $\bot_{bs}$.
- CM-PARAMS. If $ctx'_\sqcap$ is $\bot_{ctx}$, then the proof is trivial, and it cannot be $\top_{ctx}$. Thus, suppose $ctx'_\sqcap = \lessdot\ (bs_3)\ \ldots \gtrdot$. It must be that all of:

$$\begin{array}{rcl} bs_3 & \sqsubseteq & bs_1 \\ bs_3 & \sqsubseteq & bs_2 \\ ctx'_\sqcap[\texttt{self}] & \sqsubseteq & ctx_1[\texttt{self}] \\ ctx'_\sqcap[\texttt{self}] & \sqsubseteq & ctx_2[\texttt{self}] \\ \forall var: \ ctx'_\sqcap[var] & \sqsubseteq & ctx_1[var] \\ \forall var: \ ctx'_\sqcap[var] & \sqsubseteq & ctx_2[var] \end{array}$$

Since meet is correct for block specifications, $bs_3 \sqsubseteq bs_1 \sqcap bs_2$. By the inductive assumption, both:

$$\begin{array}{rcl} ctx'_\sqcap[\texttt{self}] & \sqsubseteq & ctx_1[\texttt{self}] \sqcap ctx_2[\texttt{self}] \\ \forall var: \ ctx'_\sqcap[var] & \sqsubseteq & ctx_1[var] \sqcap ctx_2[var] \end{array}$$

Thus all of the conditions are met to use rule CO-PARAMS, and $ctx'_\sqcap \sqsubseteq ctx_\sqcap$.

□

Lemma 6.20 (Comparison of Flow Positions is Reflexive)**.** For any flow position f, $f \sqsubseteq f$.

Proof. For each kind of flow position f may be, there is a rule showing that $f \sqsubseteq f$. □

Lemma 6.21 (Comparison of Flow Positions is Transitive)**.** For any three flow positions f_1, f_2, and f_3, where $f_1 \sqsubseteq f_2$ and $f_2 \sqsubseteq f_3$, it must be that $f_1 \sqsubseteq f_3$.

Proof. Consider each way that it may be derived that $f_1 \sqsubseteq f_2$:

- FO-TOP. Then $f_2 = \top_{fp}$. It must also be that $f_3 = \top_{fp}$, because otherwise it is impossible that $f_2 \sqsubseteq f_3$. Thus by FO-TOP, $f_1 \sqsubseteq f_3$.
- FO-BOTTOM. Then $f_1 = \bot_{fp}$. By FO-BOTTOM again, $f_1 \sqsubseteq f_3$.
- FO-VAR. Then:

$$\begin{aligned} f_1 &= [\!:\ V\ var\ :\!]_{ctx_1} \\ f_2 &= [\!:\ V\ var\ :\!]_{ctx_2} \\ ctx_1 &\sqsubseteq ctx_2 \end{aligned}$$

  Consider each possible way to justify that $f_2 \sqsubseteq f_3$:

  - FO-TOP. Then $f_3 = \top_{fp}$. By FO-TOP, $f_1 \sqsubseteq f_3$.
  - FO-VAR. Then $f_3 = [\!:\ V\ var\ :\!]_{ctx_3}$ where $ctx_2 \sqsubseteq ctx_3$. Since comparison of contexts is transitive, it must be that $ctx_1 \sqsubseteq ctx_3$. Thus by FO-VAR, $f_1 \sqsubseteq f_3$.
  - FO-SUM-R Then $f_3 = [\!:\ \Sigma fs_3\ :\!]$ and there is some $f_3' \in fs_3$ such that $f_2 \sqsubseteq f_3'$. One of the above rules must have been used to justify $f_2 \sqsubseteq f_3'$, and thus by the arguments given above it must be that $f_1 \sqsubseteq f_3'$. Thus, by FO-SUM-R, $f_1 \sqsubseteq f_3$.

- FO-METH. Likewise.
- FO-SUM-L. Then $f_1 = [\!:\ \Sigma fs_1\ :\!]$ for some fs_1, and for every $f_1' \in fs_1$, it must be that $f_1' \sqsubseteq f_2$. If f_2 is not a sum flow position, then one of the above arguments will show that for each $f_1' \in fs_1$, $f_1' \sqsubseteq f_3$, and thus by FO-SUM-L, $f_1 \sqsubseteq f_3$.

  Thus, suppose $f_2 = [\!:\ \Sigma fs_2\ :\!]$. Consider each $f_1' \in fs_1$ in turn. To justify that $f_1' \sqsubseteq f_2$, rule FO-SUM-R must have been used. Thus there must be some $f_2' \in fs_2$ such that $f_1' \sqsubseteq f_2'$. Now, consider each way it may have been justified that $f_2 \sqsubseteq f_3$:

  - FO-TOP. Then $f_3 = \top_{fp}$, and $f_1' \sqsubseteq f_3$.
  - FO-SUM-L. Then every element of fs_2, including f_2', is subsumed by f_3. Thus, by one of the above arguments, $f_1' \sqsubseteq f_3$.
  - FO-SUM-R. Then $f_3 = [\!:\ \Sigma fs_3\ :\!]$, and there must be some $f_3' \in fs_3$ such that $f_2 \sqsubseteq f_3'$. To justify that $f_2 \sqsubseteq f_3'$, it is only possible to use FO-SUM-L. Thus, there must be a $f_2' \in fs_2$ such that $f_2' \sqsubseteq f_3'$. By one of the above arguments, it must also be that $f_1' \sqsubseteq f_3'$. By FO-SUM-R, it must also be that $f_1' \sqsubseteq f_3$.

  In all cases, $f_1' \sqsubseteq f_3$. Since this argument holds for all $f_1' \in fs_1$, one may use FO-SUM-L to show that $f_1 \sqsubseteq f_3$.

- FO-SUM-R. Then $f_2 = [\!:\ \Sigma fs_2\ :\!]$ and there is some $f_2' \in fs_2$ such that $f_1 \sqsubseteq f_2'$. Since $f_2 \sqsubseteq f_3$, it is straightforward to show that $f_2' \sqsubseteq f_3$. Thus, by one of the arguments given previously, it must also be that $f_1 \sqsubseteq f_3$.

□

Lemma 6.22 (Comparison of Flow Positions is Antisymmetric)**.** For any flow positions f_1 and f_2 where both $f_1 \sqsubseteq f_2$ and $f_2 \sqsubseteq f_1$, it must be that $f_1 = f_2$.

Proof. The proof directly parallels the proof of antisymmetric comparison of types and contexts. □

Lemma 6.23 (Join of Flow Positions is Complete)**.** For any flow positions f_1 and f_2, there is a flow position $f_\sqcup$ such that $f_1 \sqcup f_2 = f_\sqcup$.

Proof. The proof by cases is straightforward. □

Lemma 6.24 (Meet of Flow Positions is Complete)**.** If f_1 and f_2 are any flow positions, then there is a flow position $f_\sqcap$ such that $f_1 \sqcap f_2 = f_\sqcap$.

Proof. The proof is by induction on the construction of f_1 and f_2. The proof is straightforward. □

Lemma 6.25 (Join of Flow Positions is Correct)**.** For any flow positions f_1, f_2, and $f_\sqcup$, where $f_1 \sqcup f_2 = f_\sqcup$, $f_\sqcup$ is the least upper bound of f_1 and f_2. That is, $f_1 \sqsubseteq f_\sqcup$, $f_2 \sqsubseteq f_\sqcup$, and for any $f'_\sqcup$ such that $f_1 \sqsubseteq f'_\sqcup$ and $f_2 \sqsubseteq f'_\sqcup$, it must be that $f_\sqcup \sqsubseteq f'_\sqcup$.

Lemma 6.26 (Meet of Flow Positions is Correct)**.** For any flow positions f_1, f_2, and $f_\sqcap$, where $f_1 \sqcap f_2 = f_\sqcap$, $f_\sqcap$ is the greatest lower bound of f_1 and f_2. That is, $f_\sqcap \sqsubseteq f_1$, $f_\sqcap \sqsubseteq f_2$, and for any $f'_\sqcap$ such that $f'_\sqcap \sqsubseteq f_1$ and $f'_\sqcap \sqsubseteq f_2$, it must be that $f'_\sqcap \sqsubseteq f_\sqcap$.

Proof. The proofs for these two lemmas are directly parallel to those for the analogous lemmas for types. □

6.11 Properties of cpasplit

This section proves that **cpasplit** provides decompositions with various desirable properties. The first theorem shows that the **cpasplit** of a data-flow structure includes all of the same semantic structures. That is, all of the data-flow structures defined in this chapter are abstractions of concrete semantic structures, and the lemma shows that **cpasplit** does not lose any of those concrete structures when it decomposes one abstract structure into a number of smaller abstract structures.

Theorem 3 (Full Inclusiveness of Interpretations of **cpasplit**)**.** *Any object obj that is a member of a type t is also a member of some element of* **cpasplit**(t)*. Likewise, any activation act that is matched by context c is also matched by some member of* **cpasplit**(c)*. Finally, for any object obj that is in flow position f in configuration cfg, the object is also in one of the flow positions* **cpasplit**(f) *in cfg.*

Proof. The proof is by induction over the structure of t, c, or f. First consider t, an arbitrary type.

It cannot be that $t = \bot$, because no object is a member of $\bot$. If t is a class type, a selector type, or $\top$, then **cpasplit**$(t) = \{t\}$ and one can choose t itself as the simple type of which *obj* is a member. If t is a block type $B[\![b]\!]_c$, then *obj* is a block whose activation is matched by c. By the inductive assumption, there must be a context $c' \in$ **cpasplit**(c) that also matches *obj*'s activation.

If t is a sum type Σts, then *obj* must be a member of t' for some $t' \in ts$. By the inductive assumption, *obj* must also be a member of some type in **cpasplit**(t'). That type is an element of ts' as defined in Figure 6.13. Either that type, or a type that subsumes it, is a member of **remove_redundancies**(ts'), and *obj* is a member of whichever type that is.

This concludes all cases for the type t. Now consider an arbitrary context c and any activation *act* that it matches. If $c = \top_{ctx}$ or $c = \bot_{ctx}$ then **cpasplit**$(c) = \{c\}$ and one can choose c itself as the simple context which matches *act*.

That leaves the case where c is a parameters context:

$$< (bs)\ \mathtt{self} : t_s, p_1 : t_1, \ldots, p_n : t_n >$$

Since c matches *act*, the receiver of *act* must be a member of t_s, and each parameter p_i of *act* must be a member of the corresponding t_i. By the inductive assumption, the receiver must be a member of some element t'_s of **cpasplit**(t_s), and each parameter p_i must be an element of some element t'_i of **cpasplit**(t_i). Combining those elements yields the following context which must include *act*:

$$< (bs)\ \texttt{self} : t'_s, p_1 : t'_1, \ldots, p_n : t'_n >$$

By the definition of **cpasplit**, either this context or a context which subsumes it is in **cpasplit**(c).

Finally, the proofs for decomposition of f, an arbitrary flow position, directly parallel those for types and are omitted. □

The next theorem is similar to the previous, except that it applies to abstract data-flow structures instead of concrete semantic structures.

Theorem 4 (Full Inclusiveness of **cpasplit**). *If one simple data-flow object is $\sqsubseteq$ another data-flow object df, then it is also $\sqsubseteq$ one of the elements of* **cpasplit**(*df*). *That is:*

1. *If t_s is a simple type and $t_s \sqsubseteq t$, then $\exists t' \in$* **cpasplit**(t) : $t_s \sqsubseteq t'$.
2. *If c_s is a simple context and $c_s \sqsubseteq c$, then $\exists c' \in$* **cpasplit**(c) : $c_s \sqsubseteq c'$.
3. *If f_s is a simple flow position and $f_s \sqsubseteq f$, then $\exists f' \in$* **cpasplit**(f) : $f_s \sqsubseteq f'$.

Proof. First, let us dismiss some trivial cases. If $t_s = \bot$, then one can choose any $t' \in$ **cpasplit**(t) at all. If $t_s = \top$, then it must be that $t = \top$ and thus that **cpasplit**(t) = $\{\top\}$. Choose $t' = \top$. The cases where c_s or f_s are their respective $\top$ and $\bot$ elements are proven similarly.

Ignoring these cases, the proof is by induction on the structure of the data-flow structure that **cpasplit** is applied to. If t is $\top$, $\bot$, a class type, or a selector type, then **cpasplit**(t) = $\{t\}$, and one can choose $t' = t$. If t is a block type $B\llbracket b\rrbracket_c$, then t_s must also be some block type $B\llbracket b\rrbracket_{c_s}$ where $c_s \sqsubseteq c$. By the inductive assumption, there is some $c' \in$ **cpasplit**(c) such that $c_s \sqsubseteq c'$. By the definition of **cpasplit**, either $B\llbracket b\rrbracket_{c'} \in$ **cpasplit**(t), in which case we can choose $t' = B\llbracket b\rrbracket_{c'}$, or **cpasplit**($t$) must include some type which is $\sqsupseteq B\llbracket b\rrbracket_{c'}$, in which case we can choose that type for t'.

If t is a sum type Σts, then since $t_s \sqsubseteq t$ it must be that there is a $t'' \in ts$ such that $t_s \sqsubseteq t''$. By the induction hypothesis, there is a $t''' \in$ **cpasplit**(t'') such that $t_s \sqsubseteq t'''$. By the definition of **cpasplit**, either **cpasplit**(t) must include t''', in which case we can choose $t' = t'''$, or it must include some element that is $\sqsupseteq t'''$, in which case we can choose that element as t'.

If c is $\top_{ctx}$ or $\bot_{ctx}$, then **cpasplit**(c) = $\{c\}$, so choose $c' = c$. Otherwise, c_s and c must both be parameters contexts. All of the elements of **cpasplit**(c) are also parameters contexts, each of which has a block specification the same as c's. Thus, the block specification of c_s is $\sqsubseteq$ the block specificiation of every context in **cpasplit**(c). By the inductive assumption, there is some $t'_{\texttt{self}} \in$ **cpasplit**($c[\texttt{self}]$) such that $c_s[\texttt{self}] \sqsubseteq t'$. Likewise for each of the parameter p_i types specified by c_s: there must be a type t'_i in the decomposition of parameter p_i's type in c. One can use these to construct a parameters context whose block is the same as c's, whose self type is $t'_{\texttt{self}}$, and whose parameter types are the t'_i types. Either this context must be in **cpasplit**(c), or there must be some context in **cpasplit**(c) that is $\sqsupseteq$ of it. Either way, there is a $c' \in$ **cpasplit**(c) such that $c_s \sqsubseteq c'$.

The cases for flow positions all parallel cases proven above for types and contexts. If $f = \top_{fp}$ or $f = \bot_{fp}$ then, as with the similar case for types and contexts, one can choose $f' = f$. If f is a sum flow positions, then we can use the same approach used for sum types to find an appropriate f'. If f is a self flow position or a variable flow position, then as with block types, we can use the inductive assumption on the flow position's context to find a simple context that subsumes f_s's context, and then use that to find a flow position in the decomposition of f that subsumes f_s. □

FM-SYM
$$\frac{f_2 \sqcap f_1 = f_3}{f_1 \sqcap f_2 = f_3}$$

FM-SUBSUME
$$\frac{f_1 \sqsubseteq f_2}{f_1 \sqcap f_2 = f_1}$$

FM-VAR
$$\frac{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \mathit{ctx} \qquad \mathit{ctx} \neq \bot_{\mathit{ctx}}}{[:\ V\ \mathit{var}\ :]_{\mathit{ctx}_1} \sqcap [:\ V\ \mathit{var}\ :]_{\mathit{ctx}_2} = [:\ V\ \mathit{var}\ :]_{\mathit{ctx}_1 \sqcap \mathit{ctx}_2}}$$

FM-VAR-DIFFCTX
$$\frac{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \bot_{\mathit{ctx}}}{[:\ V\ \mathit{var}\ :]_{\mathit{ctx}_1} \sqcap [:\ V\ \mathit{var}\ :]_{\mathit{ctx}_2} = \bot_{\mathit{fp}}}$$

FM-DIFFVAR
$$\frac{\mathit{var}_1 \neq \mathit{var}_2}{[:\ V\ \mathit{var}_1\ :]_{\mathit{ctx}_1} \sqcap [:\ V\ \mathit{var}_2\ :]_{\mathit{ctx}_2} = \bot_{\mathit{fp}}}$$

FM-SELF
$$\frac{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \mathit{ctx} \qquad \mathit{ctx} \neq \bot_{\mathit{ctx}}}{[:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_1} \sqcap [:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_2} = [:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_1 \sqcap \mathit{ctx}_2}}$$

FM-SELF-DIFFCTX
$$\frac{\mathit{ctx}_1 \sqcap \mathit{ctx}_2 = \bot_{\mathit{ctx}}}{[:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_1} \sqcap [:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_2} = \bot_{\mathit{fp}}}$$

FM-SELF-DIFFMETH
$$\frac{\mathit{meth}_1 \neq \mathit{meth}_2}{[:\ S\ \mathit{meth}_1\ :]_{\mathit{ctx}_1} \sqcap [:\ S\ \mathit{meth}_2\ :]_{\mathit{ctx}_2} = \bot_{\mathit{fp}}}$$

FM-VARSELF
$$\frac{}{[:\ V\ \mathit{var}\ :]_{\mathit{ctx}_1} \sqcap [:\ S\ \mathit{meth}\ :]_{\mathit{ctx}_2} = \bot_{\mathit{fp}}}$$

FM-SUM
$$\frac{\begin{array}{c}\forall f' \in \mathit{fs} : f' \sqcap f_2 = m(f') \\ f_3 = \bigsqcup_{f' \in \mathit{fs}} m(f')\end{array}}{[:\ \Sigma\, \mathit{fs}\ :] \sqcap f_2 = f_3}$$

Figure 6.12: Meet for Flow Positions

$$\textbf{cpasplit}(\top) = \top \qquad \textbf{cpasplit}(\bot) = \bot \qquad \textbf{cpasplit}(\{\!|c|\!\}) = \{\{\!|c|\!\}\} \qquad \textbf{cpasplit}(S\,\{\!|s|\!\}) = \{S\,\{\!|s|\!\}\}$$

$$\textbf{cpasplit}(B\{\!|bs|\!\}_{ctx}) = \{\ B\{\!|bs|\!\}_{ctx'} \ \mid\ ctx' \in \textbf{cpasplit}(ctx)\ \} \qquad \frac{ts' = \bigcup_{t \in ts} \textbf{cpasplit}(t)}{\textbf{cpasplit}(\Sigma ts) = \textbf{remove\_redundancies}(ts')}$$

$$\textbf{cpasplit}(\top_{ctx}) = \top_{ctx} \qquad \textbf{cpasplit}(\bot_{ctx}) = \bot_{ctx}$$

$$\frac{\begin{array}{c} st_s = \textbf{cpasplit}(t_s) \qquad \forall i \in 1 \ldots n : st_i = \textbf{cpasplit}(t_i) \\ ctxs \ = \ \{< (bs)\ \texttt{self} : t'_s, p_1 : t'_1, \ldots, p_n : t'_n > \ \mid\ t'_s \in ts_s \ \wedge\ \forall i \in 1 \ldots n : t'_i \in st_i\} \end{array}}{\textbf{cpasplit}(< (bs)\ \texttt{self} : t_s, p_1 : t_1, \ldots, p_n : t_n > = \textbf{remove\_redundancies}(ctxs)}$$

$$\textbf{cpasplit}(\top_{fp}) = \top_{fp} \qquad \textbf{cpasplit}(\bot_{fp}) = \bot_{fp}$$

$$\textbf{cpasplit}([:\ V\ v\ :]_c) = \{[:\ V\ v\ :]_{c'} \ \mid\ c' \in \textbf{cpasplit}(c)\}$$

$$\textbf{cpasplit}([:\ S\ m\ :]_c) = \{[:\ S\ m\ :]_{c'} \ \mid\ c' \in \textbf{cpasplit}(c)\}$$

$$\frac{fs' = \bigcup_{f \in fs} \textbf{cpasplit}(f)}{\textbf{cpasplit}([:\ \Sigma\, fs\ :]) = \textbf{remove\_redundancies}(fs')}$$

Figure 6.13: The function **cpasplit**, which can be used do decompose any type, context, or flow position.

Chapter 7

Justification rules

Chapter 6 describes the data-flow judgements that **DDP** produces, but says nothing about which judgements **DDP** produces, nor about how it finds those judgements. This chapter fills in both of these gaps by describing the *justification rules* available to **DDP**.

Justification rules are specified as rules of inference, such that every judgement produced by **DDP** must be justified using only those rules. When **DDP** produces a set of judgements, those judgements are always justified by the justification rules of this chapter. Looked at in reverse, whenever **DDP** tries to solve a goal and find a judgement satisfying it, it consults the available justification rules and follows them backwards. It constructs a judgement to satisfy the goal, such that the judgement can possibly be justified using the available rules.

Each justification rule has not only a conclusion, but also a number of assumptions. For the rule to be used, all of its assumptions must be satisfied. In particular, each judgement listed in the assumptions must itself be justified by another justification rule. The result is that, in general, the full justification of a judgement is a tree of justifications. Such a tree is called a *justification tree*.

7.1 Meta-judgements

The justification rules frequently refer to two meta-judgements. First, the meta-judgement:

$$\rhd\ j$$

means that judgement j is *justified* with respect to $\mathcal{J}$ and $\mathcal{P}$. A justified judgement is locally consistent with the other judgements in $\mathcal{J}$. That is, if all of the other judgements in $\mathcal{J}$ are correct, then j must be as well.

Looking ahead to the correctness proof of Chapter 8, note that this reasoning is circular and thus not enough to ensure that a set of *justified* judgements is also a set of correct judgements. For that to be the case, the justification rules, given in this chapter, are careful to ensure that, roughly, the assumptions of each rule refer only to information about the syntax of the program or to information about previous states of execution. More precisely, any chain of justifications and assumptions must eventually refer to previous states; some individual assumptions may refer to the current execution state, but there may not be a cycle of such justifications and assumptions. By consistently arranging the justification rules that their assumptions look back in time in this fashion, the stage is set for a proof by induction over steps of execution.

The second common meta-judgement referred to is a strictly weaker claim than $\rhd\ j$. It looks like:

$$\mathit{stat} \star b \quad \rhd \quad j$$

This judgement means that judgement j *accounts for* the possible execution of statement *stat* under bindings b. It means that if j is correct in one configuration, and then *stat* executes—thus moving to a new configuration—that j will remain correct in the new configuration. The bulk of the justification rules given in this chapter are techniques for justifying accounts-for meta-judgements for different kinds of statements *stat* and data-flow judgements j. To make the stronger claim that a judgement j is justified outright, one can simple show that j accounts for all statements in the program being analyzed.

7.2 Subgoals: justification rules viewed backwards

Justification rules can be viewed in reverse as a tactic for finding a solution to a goal. Whenever **DDP** updates a goal, i.e. whenever it runs the `Update` function described in Chapter 3, it finds a justification rule whose conclusion is $\rhd\ j$ for some judgement j that is a possible answer to the goal. Then it tries to satisfy each assumption of the rule. Some assumptions can be satisfied directly by simply modifying the goal's tentative solution. Others must themselves be justified, in which case **DDP** must recursively choose another justification rule and try to justify the assumption.

Some assumptions require that $j' \in \mathcal{J}$ for some judgement j' meeting some list of constraints. In such a case, **DDP** creates a *subgoal* to find a j' meeting the required constraints. Note that justification of j is only valid so long as every j' of this kind is in $\mathcal{J}$ and justified. If any such j' is removed from $\mathcal{J}$ and replaced with a different judgement, then the justification of j must also be revisited. Thus, subgoals correspond to dependencies; a goal's tentative solution depends on its subgoals' tentative solutions.

Consider an example based on the example of Chapter 3. The initial goal is "What is `X`?", which is written formally as:

$$\mathtt{X} :_{\top} ?$$

One of the justification rules is JUST-ONE, which is as follows:

$$\text{JUST-ONE}\quad \frac{\begin{array}{c}\mathbf{meets\_min}(j) \\ \forall(stat, b) \in \mathbf{bound\_stats}(P)\ : \\ stat \star b \quad \rhd \quad j\end{array}}{\rhd\ j}$$

In order to use this rule, **DDP** must meet the rule's assumptions. The strongest assumption to be met is the one that j accounts for all statements in $\mathcal{P}$. Statements that do not modify `X` are trivial to account for; the only non-trivial ones in the example program are `X := Y` and `X := p1`. Both of these may both be accounted for using the JUST-VAR justification rule:

$$\text{T-VAR}\quad \frac{v = b[l] \qquad v' = b[l'] \qquad v' :_{\lceil c \rceil} t' \in \mathcal{J} \qquad t' \sqsubseteq t}{[l := l'] \star b \quad \rhd \quad v :_c t}$$

Ignoring issues of variable bindings, and ignoring the trivial justifications, **DDP** could use these rules to reach the following tentative justification tree:

$$\frac{\dfrac{\{\text{UndefinedObject}\} \sqsubseteq t_X}{\mathbf{meets\_min}(\mathtt{X} :_{\top} t_X)} \qquad \mathtt{Y} :_{\top} t_Y \in \mathcal{J} \qquad t_Y \sqsubseteq t_X \qquad \mathtt{p1} :_{\top} t_{p1} \in \mathcal{J} \qquad t_{p1} \sqsubseteq t_X}{\rhd\ \mathtt{X} :_{\top} t_X}$$

This justification tree has three holes in it, however: t_X, t_Y, and t_{p1}. It is easy to choose t_X once the other types are known: choose the smallest type that satisfies all of the requirements in the assumptions. To fill in the holes for t_Y and t_{p1}, **DDP** creates two subgoals, one for `Y` $:_{\top}?$ and one for `p1` $:_{\top}?$. Informally, these subgoals are read "What is Y?" and "What is p1?". When these goals are initially created, they will be given a tentative solution of $\perp$. This leads to the following justification graph, which corresponds to the state of the example execution from Figure 3.4.

$$\dfrac{\begin{array}{c}\dfrac{\{\!|\text{UndefinedObject}|\!\} \sqsubseteq \{\!|\text{UndefinedObject}|\!\}}{\textbf{meets\_min}(\texttt{X} :_\top \{\!|\text{UndefinedObject}|\!\})} \qquad \texttt{Y} :_\top \bot \in \mathcal{J} \qquad \bot \sqsubseteq \{\!|\text{UndefinedObject}|\!\} \\ \texttt{p1} :_\top \bot \in \mathcal{J} \qquad t_{p1} \sqsubseteq \{\!|\text{UndefinedObject}|\!\}\end{array}}{\rhd\ \texttt{X} :_\top \{\!|\text{UndefinedObject}|\!\}}$$

Note that the type of `X` at this stage is $\{\!|\text{UndefinedObject}|\!\}$ instead of $\bot$. For clarity, Chapter 3 ignored this **meets_min** requirement and thus left out the $\{\!|\text{UndefinedObject}|\!\}$'s from the entire example execution.

Also, note that the judgement $\texttt{X} :_\top \{\!|\text{UndefinedObject}|\!\}$ is probably not correct. This example justification only shows that the judgement is *consistent* with certain other judgements in $\mathcal{J}$, namely $\texttt{Y} :_\top \bot$ and $\texttt{p1} :_\top \bot$. These two judgements are probably not justifiable. As **DDP** progresses, it will adjust them to be justifiable, that is $\texttt{Y} :_\top t_1$ for some type t_1 and $\texttt{p1} :_\top t_2$ for some type t_2, but then $\texttt{X} :_\top \{\!|\text{UndefinedObject}|\!\}$ will no longer be justifiable and must itself be adjusted. Thus, changes ripple from judgements to other judgements depending on them, to other judgements depending on those, and so on until all judgements are justified with respect to each other.

7.3 Overall justification approach

This subsection gives several general strategies available for justifying a judgement. All of these justification rules are listed in Figure 7.1.

First, the judgement may be given some conservative value that is clearly correct regardless of how the program behaves. Such a judgement is justified with one of the rules: JUST-PRUNE-TYPE, JUST-PRUNE-TFLOW, JUST-PRUNE-FLOW, JUST-PRUNE-SEND, or JUST-PRUNE-RESP.

Second, one might show that a judgement is tautological. The rule JUST-CTX is such a rule: the type in the judgement subsumes the type that the context already presumes the variable will hold.

Finally, a judgement may account for every statement in the program and be justified by the rule JUST-ONE. The JUST-ONE rule requires that a judgement account for the possible execution of every statement in the program. Additionally, JUST-ONE requires that any judgement meets a certain minimum value. In combination, these two requirements prepare for an inductive proof of correctness. The first part requires that no matter which statement executes the judgement will remain correct, and the second part shows that the judgement is correct initially. The minimum judged values are shown in Figure 7.2. They require that any type judgement for a non-parameter includes the type of `NilObj`, because `NilObj` is the initial value automatically assigned to variable whenever a new contour is allocated that holds that variable.

Not all judgements are justified by the rules of Figure 7.1. In particular, transitive flow judgements, senders judgements, and responders judgements have their own justification rules which are described later.

7.4 Type justifications

Figure 7.3 gives several type justifications that are trivial. Most of these rules justify judgements where the only variable changed is different from the one the judgement refers to.

Figure 7.4 gives the non-trivial type justifications for all other statement types. For the most part these are straightforward. For example, T-VAR accounts for a statement $l := l'$ by the type for the variable on the left being larger than a type for the variable on the right in the same context (or more specifically, the same context after the context broadening described in section 6.9). To a first approximation, T-SELF accounts for a statement $l :=$ `self` by the type for the variable on the left including the class cone type for the class the statement appears in. To a closer approximation, T-SELF allows this type to be whittled down by the context of the type judgement.

The type justification rules for method and block invocations are T-SEND, T-SENDVAR, and T-BEVAL. Each of these reduces the justification to the justifications of two other judgements. One judgement accounts

$$\text{JUST-PRUNE-TYPE}\;\frac{}{\rhd\, v :_c \top}\qquad \text{JUST-PRUNE-FLOW}\;\frac{\textit{fp is a simple flow position}}{\rhd\, fp \to \top_{fp}}\qquad \text{JUST-PRUNE-TFLOW}\;\frac{}{\rhd\, fp \to^* \top_{fp}}\qquad \text{JUST-PRUNE-SEND}\;\frac{ss = \top_s}{\rhd\, m_{ctx} \xleftarrow{send} ss}$$

$$\text{JUST-PRUNE-RESP}\;\frac{rs = \top_r}{\rhd\, stat_{ctx} \star b \xrightarrow{send} rs}\qquad \text{JUST-ONE}\;\frac{\begin{array}{c}\textbf{meets\_min}(j)\\ \forall (stat, b) \in \textbf{bound\_stats}(P)\ :\\ stat \star b \quad \rhd \quad j\end{array}}{\rhd\, j}\qquad \text{JUST-CTX}\;\frac{c[v] \sqsubseteq t}{\rhd\, v :_c t}$$

Figure 7.1: Overall Justification Rules

$$\text{MIN-NONTYPE}\;\frac{\textit{j is not a type judgement}}{\textbf{meets\_min}(j)}\qquad \text{MIN-PARAMETER}\;\frac{\textit{v is a parameter}}{\textbf{meets\_min}(v :_c t)}\qquad \text{MIN-VAR}\;\frac{\lfloor \text{UndefinedObject} \rfloor \sqsubseteq t}{\textbf{meets\_min}(v :_c t)}$$

Figure 7.2: Minimum Requirements of Judgements

for types of the variable to be assigned when the method or block returns, while the other judgement accounts for types of the parameters of any methods or blocks that might be invoked by the statement.

Figure 7.5 gives the first group: they account for the type of the variable on the left. The three trivial justifications, T-SEND-R-TRIV, T-SENDVAR-R-TRIV, and T-BEVAL-R-TRIV, require the assigned variable to be different from the variable of the type judgement. The other three follow this pattern:

1. Find the methods or blocks that may be invoked by the statement.

2. Find the overall context that the method or block will run in, by finding types for the parameters and, for method invocations, the receiver.

3. Divide that context, in **CPA** fashion, into a number of small contexts.

4. For each combination of an invoked method or block, a return statement in that method or block, and one of the small contexts, find a type for the returned variable and require that the type of the variable being judged is a supertype of that type.

Figure 7.6 gives justifications for parameter type judgements that were not justified by the rule T-CONTEXT. Aside from the trivial justifications, the pattern for all of them is to find the statements that might invoke the relevant block and, for each of these, to find a type for the relevant argument passed by the statement.

7.5 Flow justifications

Figure 7.7 and Figure 7.8 give those justifications for flow judgements that are trivial. There are many of them; in addition to justifications based on mismatching variables, there are justifications based on the statement type not having the relevant kind of flow at all. Self flow positions only flow via `self` statements, and variable flow positions only flow via variable assignments, message sends, and returns. No flow at all happens for literal creation, class instantiation, and block creation, because these statement types only create new objects and do not move around existing objects.

T-LIT-TRIV
$$\frac{v \neq b[l]}{[l := lit] \star b \quad \rhd \quad v :_c t}$$

T-VAR-TRIV
$$\frac{v \neq b[l]}{[l := l'] \star b \quad \rhd \quad v :_c t}$$

T-NEW-TRIV
$$\frac{v \neq b[l]}{[l := \texttt{new}\ \ class] \star b \quad \rhd \quad v :_c t}$$

T-BLOCK-TRIV
$$\frac{v \neq b[l]}{[l := block] \star b \quad \rhd \quad v :_c t}$$

Figure 7.3: Trivial Type Justifications

T-LIT
$$\frac{v = b[l] \qquad t' = \mathbf{lit\_type}(lit) \sqsubseteq t}{[l := lit] \star b \quad \rhd \quad v :_c t}$$

T-SEL
$$\frac{\begin{array}{c} v = b[l] \\ selector = \text{Selector label: } l_s \ \text{ numargs: } m_s \\ t' = S\llparenthesis l_s, m_s \rrparenthesis \sqsubseteq t \end{array}}{[l := selector] \star b \quad \rhd \quad v :_c t}$$

T-VAR
$$\frac{v = b[l] \qquad v' = b[l'] \qquad v' :_{\lceil c \rceil} t' \in \mathcal{J} \qquad t' \sqsubseteq t}{[l := l'] \star b \quad \rhd \quad v :_c t}$$

T-SELF
$$\frac{v = b[l] \qquad (c[\texttt{self}] \sqcap \llparenthesis m.class \rrparenthesis^{+}) \sqsubseteq t}{[l := \texttt{self}] \star b \quad \rhd \quad v :_c t}$$

T-NEW
$$\frac{v = b[l] \qquad \llparenthesis class \rrparenthesis \sqsubseteq t}{[l := \texttt{new}\ \ class] \star b \quad \rhd \quad v :_c t}$$

T-BLOCK
$$\frac{v = b[l] \qquad B\llparenthesis block \rrparenthesis_{\lceil c \rceil} \sqsubseteq t}{[l := block] \star b \quad \rhd \quad v :_c t}$$

T-SEND
$$\frac{\begin{array}{c} [l := \mathsf{send}(l_{rcvr}, sel, l_1, \ldots, l_m)] \star b \quad R \rhd \quad v :_c t \\ [l := \mathsf{send}(l_{rcvr}, sel, l_1, \ldots, l_m)] \star b \quad S \rhd \quad v :_c t \end{array}}{[l := \mathsf{send}(l_{rcvr}, sel, l_1, \ldots, l_m)] \star b \quad \rhd \quad v :_c t}$$

T-SENDVAR
$$\frac{\begin{array}{c} [l := \texttt{sendvar}(semvarl_{rcvr}, l_{selvar}, l_1, \ldots, l_m)] \star b \quad R \rhd \quad v :_c t \\ [l := \texttt{sendvar}(semvarl_{rcvr}, l_{selvar}, l_1, \ldots, l_m)] \star b \quad S \rhd \quad v :_c t \end{array}}{[l := \texttt{sendvar}(semvarl_{rcvr}, l_{selvar}, l_1, \ldots, l_m)] \star b \quad \rhd \quad v :_c t}$$

T-BEVAL
$$\frac{[l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t \qquad [l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}{[l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad \rhd \quad v :_c t}$$

Figure 7.4: Type Justifications.

T-SEND-R-TRIV
$$\frac{v \neq b[l]}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

T-SEND-R
$$\frac{\begin{array}{c} v = b[l] \qquad stat = [\![l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)]\!] \\ stat_{ctx} \star b \xrightarrow{send} rs \ \in \ \mathcal{J} \qquad ctx = c \qquad rs = (m_1, c_1) \ldots (m_p, c_p) \\ \forall i \in 1 \ldots p : \forall\ v_{ret} \in \mathbf{ret\_vars}(m_i) : \\ \exists t' : (v_{ret} :_{c_i} t' \in \mathcal{J}) \land (t' \sqsubseteq t) \end{array}}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

T-SENDVAR-R-TRIV
$$\frac{v \neq b[l]}{[l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

T-SENDVAR-R
$$\frac{\begin{array}{c} stat = [\![l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)]\!] \\ stat_{ctx} \star b \xrightarrow{send} rs \ \in \ \mathcal{J} \qquad ctx = c \qquad rs = (m_1, c_1) \ldots (m_p, c_p) \\ \forall i \in 1 \ldots p : \forall\ v_{ret} \in \mathbf{ret\_vars}(m_i) : \\ \exists t' : (v_{ret} :_{c_i} t') \in \mathcal{J}) \land (t' \sqsubseteq t) \end{array}}{[l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

T-BEVAL-R-TRIV
$$\frac{v \neq b[l]}{[l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

T-BEVAL-R
$$\frac{\begin{array}{c} stat = [\![l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)]\!] \\ stat_{ctx} \star b \xrightarrow{send} rs \ \in \ \mathcal{J} \qquad ctx = c \qquad rs = (blk_1, bctx_1) \ldots (blk_p, bctx_p) \\ \forall i \in 1 \ldots p : blk_i.retFromMethod \lor \exists t' : (\mathbf{ret\_var}(blk_i) :_{bctx_i} t' \in \mathcal{J}) \land (t' \sqsubseteq t) \end{array}}{[l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad R \rhd \quad v :_c t}$$

Figure 7.5: Return Type from Subroutine Invocations

T-SEND-S-TRIV

$$\frac{\textit{(v is not a method parameter)}}{[l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

T-SEND-S

$$\frac{\begin{array}{c}\textit{(v is the k-th parameter of method meth)}\\ stat = [\![l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)]\!]\\ meth_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = c\\ \forall (sstat, sb, sctx) \in ss :\\ sstat \neq stat \ \vee \ sb \neq b \ \vee \ (\exists t' : (b[l_k] :_{sctx} t' \in \mathcal{J}) \wedge (t' \sqsubseteq t))\end{array}}{[l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

T-SENDVAR-S-TRIV

$$\frac{\textit{(v is not a method parameter)}}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

T-SENDVAR-S

$$\frac{\begin{array}{c}\textit{(v is the k-th parameter of method meth)}\\ stat = [\![l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)]\!]\\ meth_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = c\\ \forall (sstat, sb, sctx) \in ss :\\ sstat \neq stat \ \vee \ sb \neq b \ \vee \ (\exists t' : (b[l_k] :_{sctx} t' \in \mathcal{J}) \wedge (t' \sqsubseteq t))\end{array}}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

T-BEVAL-S-TRIV

$$\frac{\textit{(v is not a block parameter)}}{[l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

T-BEVAL-S

$$\frac{\begin{array}{c}\textit{(v is the k-th parameter of block blk)}\\ stat = [\![l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)]\!]\\ blk_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = c\\ \forall (sstat, sb, sctx) \in ss :\\ sstat \neq stat \ \vee \ sb \neq b \ \vee \ (\exists t' : (b[l_k] :_{sctx} t' \in \mathcal{J}) \wedge (t' \sqsubseteq t))\end{array}}{[l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad S \rhd \quad v :_c t}$$

Figure 7.6: Parameter Types after Subroutine Invocations

Figure 7.9 gives the flow justifications that are non-trivial. F-SELF and F-VAR are straightforward. The three send justifications F-SEND, F-SENDVAR, and F-BEVAL simply divide the justification into smaller justifications:

- Justification that the judgement accounts for the flow of the receiver:

  $$[stat] \star b \quad \texttt{self} \rhd \quad [:\ V\ v\ :]_c \to f$$

  This meta-judgement claims that the flow judgement $[:\ V\ v\ :]_c \to f$ accounts for the flow from the receiver of *stat* into the method receiver of any method that may be invoked by *stat*.

- Justification that the judgement accounts for the flow of each parameter:

  $$stat \star b \quad i \rhd \quad [:\ V\ v\ :]_c \to f$$

  This meta-judgement claims that the flow judgement accounts for flow into the *i*th parameter of any invoked method by *stat*.

- Justification that the judgement accounts for flow from the method or block back to the statement:

  $$stat \star b \quad < ms > \rhd \quad [:\ V\ v\ :]_c \to f$$

  This meta-judgement claims that the judgement accountss for flow from the return of method *ms* into the assigned variable of *stat*.

Figure 7.10, Figure 7.11, and Figure 7.12 give justifications for flow into a method or block. They follow exactly the same pattern as the justification for types of method or block parameters. Figure 7.13, Figure 7.14, and Figure 7.15 give justifications for flow out of a method and block through returned values.

Figure 7.16 gives the justification rule for transitive flow judgements. It insists that for some decomposition of the target into a number of components, there are simple flow judgements from each component back into the target. Typically, f' will be a sum flow position $[:\ \Sigma\, fs\ :]$, and the implementation will choose a decomposition of f' into the elements of *fs*.

7.6 Responders justifications

There are three responders justification rules, corresponding to the three statement types that can invoke a subroutine. They are R-SEND, R-SENDVAR, and R-BEVAL, and they are listed in Figure 7.17.

In each case, the justification relies on having a type for each argument, receiver, block variable, and selector variable that is present in the statement. These types are used to predict which methods or blocks will be invoked by the statement when it executes. In R-SEND, the combination of the type of the receiver and the selector that is present limit the number of methods that may be invoked. In R-SENDVAR, the possible selectors are found using the **possible_selectors** function on the type of the selector variable; this function assumes that its argument has a finite number of selector types included, and it returns the list of selectors corresponding to those types. In R-BEVAL, the possible responding blocks are similarly found by using the **possible_blocks** function. For both R-SENDVAR and R-BEVAL, if there are not a finite number of selectors or blocks, then the function may not be used and thus the justification may not be used. In such a case the responders judgement can only justified with JUST-PRUNE-RESP.

Once the responding methods or blocks are found, the argument types are used to find the contexts under which the method can execute. Those contexts are split into multiple smaller contexts using **cpasplit**. Finally, the responders set is required to include each possible pair of a small context and a method or block.

F-LIT
$$\frac{}{[l := \mathit{literal}] \star b \quad \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-VAR-TRIV
$$\frac{v \neq b[l]}{[l' := l] \star b \quad \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-NEW
$$\frac{}{[l := \texttt{new}\ \ \mathit{class}] \star b \quad \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-SELF-TRIV
$$\frac{}{[l := \texttt{self}] \star b \quad \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

Figure 7.7: Trivial Flow Justifications for Variable Flow Positions

F-SELF-TRIV0
$$\frac{m \neq b[\texttt{method}]}{[l := \texttt{self}] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV1
$$\frac{}{[l := \mathit{literal}] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV2
$$\frac{}{[l := l'] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV3
$$\frac{}{[l := \texttt{new}\ \ \mathit{class}] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV4
$$\frac{}{[l := \mathit{block}] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV5
$$\frac{}{[l := \texttt{send}(l_r, \mathit{sel}, l_1 \ldots l_m)] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV6
$$\frac{}{[l := \texttt{send}(l_r, \mathit{selvar}, l_1 \ldots l_m)] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

F-SELF-TRIV7
$$\frac{}{[l := \texttt{beval}(l_b, l_1 \ldots l_m)] \star b \quad \rhd \quad [:\ S\ \ m\ :]_c \rightarrow f}$$

Figure 7.8: Trivial Flow Justifications for Self Flow Positions

F-VAR

$$\frac{v = b[l] \qquad v' = b[l'] \qquad [:\ V\ v'\ :]_{\lceil c \rceil} \sqsubseteq f}{[l' := l] \star b \quad \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-SELF

$$\frac{v = b[l] \qquad m = b[\mathtt{method}] \qquad [:\ V\ v\ :]_{\lceil c \rceil} \sqsubseteq f}{[l := \mathtt{self}] \star b \quad \triangleright \quad [:\ S\ \ m\ :]_c \to f}$$

F-SEND

$$\frac{\begin{array}{c} stat = l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m) \\ [stat] \star b \ \ \mathtt{self} \triangleright \ \ [:\ V\ v\ :]_c \to f \qquad \forall i \in 1 \ldots m :\ stat \star b \ \ i \triangleright \ \ [:\ V\ v\ :]_c \to f \\ \forall ms \in \textbf{method\_specs}(\mathcal{P}) :\ stat \star b \quad < ms > \triangleright \ \ [:\ V\ v\ :]_c \to f \end{array}}{[l_l := \mathtt{send}(l_{rcvr}, sel, l_1, \ldots, l_m)] \star b \quad \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-SENDVAR

$$\frac{\begin{array}{c} stat = l_l := \mathtt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m) \\ [stat] \star b \ \ \mathtt{self} \triangleright \ \ [:\ V\ v\ :]_c \to f \qquad \forall i \in 1 \ldots m :\ stat \star b \ \ i \triangleright \ \ [:\ V\ v\ :]_c \to f \\ \forall ms \in \textbf{method\_specs}(\mathcal{P}) :\ stat \star b \quad < ms > \triangleright \ \ [:\ V\ v\ :]_c \to f \end{array}}{[l_l := \mathtt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-BEVAL

$$\frac{\begin{array}{c} stat = l_l := \mathtt{beval}(l_{blockvar}, l_1 \ldots l_m) \qquad \forall i \in 1 \ldots m :\ stat \star b \ \ i \triangleright \ \ [:\ V\ v\ :]_c \to f \\ \forall bs \in \textbf{block\_specs}(\mathcal{P}) :\ stat \star b \quad < bs > \triangleright \ \ [:\ V\ v\ :]_c \to f \end{array}}{[l_l := \mathtt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad \triangleright \quad [:\ V\ v\ :]_c \to f}$$

Figure 7.9: Non-Trivial Flow Justifications

F-SEND-PARAM-TRIV

$$\frac{b[l_k] \neq v}{[l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad k \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-SEND-PARAM

$$\frac{\begin{array}{c} [\![l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)]\!]_{ctx} \star b \xrightarrow{send} rs \in \mathcal{J} \qquad rs = (m_1, ctx_1) \ldots (m_p, ctx_p) \\ \forall i \in 1 \ldots p :\ [:\ V\ m_i.params[k]\ :]_{ctx_i} \sqsubseteq f \end{array}}{[l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad k \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-SEND-SELF-TRIV

$$\frac{rcvr \neq v}{[l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \mathtt{self} \triangleright \quad [:\ V\ v\ :]_c \to f}$$

F-SEND-SELF

$$\frac{\begin{array}{c} [\![l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)]\!]_{ctx} \star b \xrightarrow{send} rs \in \mathcal{J} \qquad rs = (m_1, ctx_1) \ldots (m_p, ctx_p) \\ \forall i \in 1 \ldots p :\ [:\ S\ \ m_i\ :]_{ctx_i} \sqsubseteq f \end{array}}{[l_l := \mathtt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \mathtt{self} \triangleright \quad [:\ V\ v\ :]_c \to f}$$

Figure 7.10: Flow into Method Invocation

F-SENDVAR-PARAM-TRIV

$$\frac{v \neq b[l_k]}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad k \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

F-SENDVAR-PARAM

$$\frac{\begin{array}{c}\llbracket l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)\rrbracket_{ctx} \star b \xrightarrow{send} rs \in \mathcal{J} \qquad rs = (m_1, ctx_1) \ldots (m_p, ctx_p) \\ \forall i \in 1 \ldots p :\ [\!:\, S\ m_i.params[k] \,:\!]_{ctx_i} \sqsubseteq f\end{array}}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad k \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

F-SENDVAR-SELF-TRIV

$$\frac{v \neq b[l_{rcvr}]}{[l_l := \texttt{send}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad \texttt{self} \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

F-SENDVAR-SELF

$$\frac{\begin{array}{c}\llbracket l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)]\rrbracket_{ctx} \star b \xrightarrow{send} rs \in \mathcal{J} \qquad rs = (m_1, ctx_1) \ldots (m_p, ctx_p) \\ \forall i \in 1 \ldots p :\ [\!:\, S\ m_i \,:\!]_{ctx_i} \sqsubseteq f\end{array}}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad \texttt{self} \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

Figure 7.11: Flow into `sendvar` Statements

F-BEVAL-PARAM-TRIV

$$\frac{v \neq b[l_k]}{[l_l := \texttt{beval}(l_{block}, l_1 \ldots l_m)] \star b \quad k \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

F-BEVAL-PARAM

$$\frac{\begin{array}{c} v_k = v \qquad \forall i \in 1 \ldots m : v_i = b[l_i] \\ stat = l_l := \texttt{beval}(l_{block}, l_1 \ldots l_m) \qquad stat_{ctx} \star b \xrightarrow{send} rs \in \mathcal{J} \qquad rs = (blk_1, bctx_1) \ldots (blk_p, bctx_p) \\ \forall i \in 1 \ldots p : [\!:\, V\ blk_i.parms[k] \,:\!]_{bctx_i} \sqsubseteq f\end{array}}{[l_l := \texttt{beval}(l_{block}, l_1 \ldots l_m)] \star b \quad k \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

Figure 7.12: Flow into `beval` Statements

F-RETURN-SEND-BADVAR

$$\frac{meth = \textbf{lookup\_meth\_spec}_{\mathcal{P}}(m_{called}) \qquad v \notin \textbf{ret\_vars}(meth)}{[l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad < m_{called} > \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

F-RETURN-SEND

$$\frac{\begin{array}{c} stat = l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m) \\ meth_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = \lceil c \rceil \qquad meth = m_{called} \\ \forall (sstat, sb, sctx) \in ss : \\ stat \neq sstat \ \vee\ sb \neq b \ \vee\ [\!:\, V\ b[l] \,:\!]_{\lceil sctx \rceil} \sqsubseteq f\end{array}}{[l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad < m_{called} > \rhd \quad [\!:\, V\ v \,:\!]_c \rightarrow f}$$

Figure 7.13: Flow from methods into `send` statements

F-RETURN-SENDVAR-BADVAR

$$\frac{meth = \mathbf{lookup\_meth\_spec}_{\mathcal{P}}(m_{called}) \qquad v \notin \mathbf{ret\_vars}(meth)}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad < m_{called} > \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-RETURN-SENDVAR

$$\frac{\begin{array}{c} stat = l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m) \\ meth_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = \lceil c \rceil \qquad meth = m_{called} \\ \forall (sstat, sb, sctx) \in ss : \\ stat \neq sstat \ \vee \ sb \neq b \ \vee \ [:\ V\ b[l]\ :]_{\lceil sctx \rceil} \sqsubseteq f \end{array}}{[l_l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad < m_{called} > \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

Figure 7.14: Flow from methods into `sendvar` statements

F-BRETURN-BEVAL-BADVAR

$$\frac{block = \mathbf{lookup\_block\_spec}_{\mathcal{P}}(b_{invoked}) \qquad block.returns \neq v}{[l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad < b_{invoked} > \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-BRETURN-BEVAL-METHRET

$$\frac{block = \mathbf{lookup\_block\_spec}_{\mathcal{P}}(b_{invoked}) \qquad block.retFromMethod}{[l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad < b_{invoked} > \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

F-BRETURN-BEVAL

$$\frac{\begin{array}{c} stat = l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m) \\ blk_{ctx} \xleftarrow{send} ss \in \mathcal{J} \qquad ctx = \lceil c \rceil \qquad blk = b_{invoked} \\ \forall (sstat, sb, sctx) \in ss : \\ stat \neq sstat \ \vee \ sb \neq b \ \vee \ [:\ V\ b[l]\ :]_{\lceil sctx \rceil} \sqsubseteq f \end{array}}{[l_l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad < b_{invoked} > \rhd \quad [:\ V\ v\ :]_c \rightarrow f}$$

Figure 7.15: Flow from blocks into `beval` statements

F-TRANS

$$\frac{\begin{array}{c} f \sqsubseteq f' \\ f_1 \sqcup f_2 \sqcup \cdots \sqcup f_p = f' \\ \forall i \in 1 \ldots p : \ \exists f_i' : \ f_i \rightarrow f_i' \in \mathcal{J} \ \wedge \ f_i' \sqsubseteq f' \end{array}}{\rhd f \rightarrow^* f'}$$

Figure 7.16: Transitive Flow Judgements

7.7 Senders justifications

Senders judgements are justified indirectly, using JUST-ONE. A justified senders judgement must account for every statement in the program.

Figure 7.18 gives various trivial ways that a judgement may account for a statement. S-SELF, S-LIT, S-SEL, S-VAR, S-NEW, and S-BLOCK rely on the statement not invoking any subroutine at all. S-SEND-TRIV, S-SENDVAR-TRIV, and S-BEVAL-TRIV rely on the statement invoking the wrong kind of subroutine, e.g. the fact that `beval` statements always invoke blocks and not methods.

The non-trivial justifications are shown in Figure 7.19. For `send` statements, the statement may be left out of the senders set if either the selector does not match (S-SEND-BADSELECTOR), or if there is a type for the receiver such that the method cannot be executed (S-SEND-BADRECV). Otherwise, the statement must be included in the senders set (S-SEND).

For `sendvar` and `beval` statements, a different approach is taken. The flow is traced forward for the relevant selector or block. Only if the flow reaches the statement must the statement be included in the senders set; if the flow does reach the statement, the statement is added to the senders set with no further inquiry. The justification rules that implement this approach are S-SENDVAR and S-BEVAL.

In S-SENDVAR, flow is traced forward from each statement creating a selector object for the relevant selector. The function **flow_select** is then used to extract the portion of the reached flow position that matches the variable from which the `sendvar` statement is reading its selector. Note that if there is no statement instantiating a selector object for the relevant selector, then one can choose $f_{sel} = \perp_{ctx}$ and thus use S-SENDVAR to reject all `sendvar` statements as potential senders. In S-BEVAL, the function **blk_stat** is used to find the statement that creates the block, and then **flow_select** is used to find that portion of the reached flow position that matches the variable the `beval` statement reads its block from.

R-SEND

$$\frac{\begin{array}{c} rcvr = b[l_{rcvr}] \quad \forall i \in 1\ldots m : v_i = b[l_i] \quad rcvr :_{\lceil ctx \rceil} t_{rcvr} \in \mathcal{J} \quad \forall i \in 1\ldots m : v_i :_{\lceil ctx \rceil} t_i \in \mathcal{J} \\ (m_1, \ldots, m_n) = \mathbf{lookup}^*{}_{\mathcal{P}}(t_{rcvr}, sel) \quad \forall i \in 1\ldots n : c_i = <(m_i)\ \texttt{self} = t_{rcvr},\ \ldots,\ m_i.parm[m] = t_m > \\ \forall i \in 1\ldots n : (c_{(i,1)}, \ldots, c_{(i,p_i)}) = \mathbf{cpasplit}(c_i) \quad \forall i \in 1\ldots n : \forall j \in 1\ldots p_i : (m_i, c_{(i,j)}) \in rs \end{array}}{\rhd\ [\![l := \texttt{send}(l_{rcvr}, selector, l_1 \ldots l_m)]\!]_{ctx} \star b \xrightarrow{send} rs}$$

R-SENDVAR

$$\frac{\begin{array}{c} rcvr = b[l_{rcvr}] \quad selvar = b[l_{semvar}] \quad \forall i \in 1\ldots m : v_i = b[l_i] \quad rcvr :_{\lceil ctx \rceil} t_{rcvr} \in \mathcal{J} \\ selvar :_{\lceil ctx \rceil} t_{sel} \in \mathcal{J} \quad \forall i \in 1\ldots m : v_i :_{\lceil ctx \rceil} t_i \in \mathcal{J} \quad (sel_1, \ldots, sel_q) = \mathbf{possible\_selectors}(t_{sel}) \\ (m_1, \ldots, m_n) = \mathbf{append}(\mathbf{lookup}^*{}_{\mathcal{P}}(t_{rcvr}, sel_1), \ldots, \mathbf{lookup}^*{}_{\mathcal{P}}(t_{rcvr}, sel_q)) \\ \forall i \in 1\ldots n : c_i = <(m_i)\ \texttt{self} = t_{rcvr},\ \ldots,\ m_i.parm[m] = t_m > \\ \forall i \in 1\ldots n : (c_{(i,1)}, \ldots, c_{(i,p_i)}) = \mathbf{cpasplit}(c_i) \quad \forall i \in 1\ldots n : \forall j \in 1\ldots p_i : (m_i, c_{(i,j)}) \in rs \end{array}}{\rhd\ [\![l := \texttt{sendvar}(l_{rcvr}, selvar, l_1 \ldots l_m)]\!]_{ctx} \star b \xrightarrow{send} rs}$$

R-BEVAL

$$\frac{\begin{array}{c} blockvar = b[l_{blockvar}] \quad \forall i \in 1\ldots m : v_i = b[l_i] \quad blockvar :_{\lceil ctx \rceil} t_{blocks} \in \mathcal{J} \\ \forall i \in 1\ldots m : v_i :_{\lceil ctx \rceil} t_i \in \mathcal{J} \quad (B\{\!|blk_1|\!\}_{bctx_1}, \ldots, B\{\!|blk_n|\!\}_{bctx_n}) = \mathbf{possible\_blocks}(t_{blocks}) \\ \forall i \in 1\ldots n : c_i = <(blk_i)\ blk_i.parm[1] = t_1,\ \ldots,\ blk_i.parm[m] = t_m > \\ \forall i \in 1\ldots n : (c_{(i,1)}, \ldots, c_{(i,p_i)}) = \mathbf{cpasplit}(c_i) \\ \forall i \in 1\ldots n :\ \forall j \in 1\ldots p_i :\ (blk_i, c_{(i,j)}) \in rs \end{array}}{\rhd\ [\![l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)]\!]_{ctx} \star b \xrightarrow{send} rs}$$

Figure 7.17: Responders Justifications

S-SELF

$$\frac{}{[l := \texttt{self}] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-LIT

$$\frac{}{[l := literal] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-VAR

$$\frac{}{[l := l'] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-NEW

$$\frac{}{[l := \texttt{new}\ \ cname] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-BLOCK

$$\frac{}{[l := block] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-SEND-TRIV

$$\frac{(bs \text{ does not specify a method})}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-SENDVAR-TRIV

$$\frac{(bs \text{ does not specify a method})}{[l := \texttt{sendvar}(l_{rcvr}, selvar, l_1 \ldots l_m)] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

S-BEVAL-TRIV

$$\frac{(bs \text{ specifies a method})}{[l := \texttt{beval}(l_{block}, l_1 \ldots l_m)] \star b \quad \rhd \quad bs_{ctx} \xleftarrow{send} ss}$$

Figure 7.18: Trivial Senders Justifications

S-SEND-BADSELECTOR

$$\frac{sel \neq bs.selector}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \triangleright \quad bs_{ctx} \xleftarrow{send} ss}$$

S-SEND-BADRECV

$$\frac{\begin{array}{c} rcvr = b[l_{rcvr}] \qquad rcvr :_{\top_{ctx}} t_{rcvr} \in \mathcal{J} \\ b.method \notin \mathbf{lookup}^*{}_{\mathcal{P}}(t_{rcvr}, sel) \end{array}}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \triangleright \quad bs_{ctx} \xleftarrow{send} ss}$$

S-SEND

$$\frac{\begin{array}{c} stat = l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m) \\ (stat, b, \top_{ctx}) \in ss \end{array}}{[l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)] \star b \quad \triangleright \quad bs_{ctx} \xleftarrow{send} ss}$$

S-SENDVAR

$$\frac{\begin{array}{c} stat = l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m) \\ rcvr = b[l_{rcvr}] \qquad selvar = b[l_{selvar}] \\ \forall (l_s := sel, b'') \in \mathbf{bound\_stats}(\mathcal{P}) : \exists f'' : \\ ([: V\ \mathbf{static\_bindings}(b'')[l_s]\ :]_{\top_{ctx}} \rightarrow^* f'' \in \mathcal{J}) \wedge (f'' \sqsubseteq f_{sel}) \\ \forall [: V\ selvar\ :]_{c'} \in \mathbf{flow\_select}(f_{sel}, selvar) : \exists (rcvr :_{\lceil c' \rceil} t_r \in \mathcal{J}) : \\ (m_{called} \notin (\mathbf{lookup}^*{}_{\mathcal{P}}(t_r, sel))) \vee ((stat, b, c') \in ss) \end{array}}{[l := \texttt{sendvar}(l_{rcvr}, l_{selvar}, l_1 \ldots l_m)] \star b \quad \triangleright \quad bs_{ctx} \xleftarrow{send} ss}$$

S-BEVAL

$$\frac{\begin{array}{c} stat = l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m) \\ blockvar = b[l_{blockvar}] \\ blk\_inst := blk = \mathbf{blk\_stat}(b') \qquad [: V\ blk\_inst\ :]_{\lceil ctx \rceil} \rightarrow^* f_b \in \mathcal{J}^\star \\ ([: V\ blockvar\ :]_{ctx_1},\ \ldots,\ [: V\ blockvar\ :]_{ctx_p}) = \mathbf{flow\_select}(f_b, blockvar) \\ \forall i \in 1 \ldots p : (stat, b, ctx_i) \in ss \end{array}}{[l := \texttt{beval}(l_{blockvar}, l_1 \ldots l_m)] \star b \quad \triangleright \quad bs_{ctx} \xleftarrow{send} ss}$$

Figure 7.19: Non-Trivial Senders Justifications

Chapter 8

Correctness of DDP

8.1 Overview

This chapter proves the following theorem:

Theorem 5 (Correctness of **DDP**)**.** *If a set of judgements $\mathcal{J}$ is justified with respect to a program $\mathcal{P}$, then $\mathcal{J}$ is correct for $\mathcal{P}$.*

Essentially this theorem verifies that the voluminous and subtle justification rules of Chapter 7 result in a set of judgements that are correct according to the straightforward definitions of Chapter 6 and under the straightforward semantics of Chapter 5. This choice applies systematic mathematical analysis to the portion of the problem where subtle errors are easy to make and where mathematical analysis is especially effective.

Some portions of the algorithm's correctness are left unverified by this theorem. In particular, it does not verify that the dependency-driven worklist portion of the algorithm generates a set of justified judgements. However, that portion of the algorithm is similar enough to proven worklist algorithms for intraprocedural data-flow analysis [5] that we believe the reader will be confident the algorithm generates sets of justified judgements even without a proof.

Likewise, the pruning algorithm is not addressed in this document's mathematical analysis. So long as the chosen pruning algorithm limits its activities to pruning per se, i.e. to giving solutions to goals that are conservative enough to be correct without requiring any subgoals, we trust the reader will believe that overall algorithm produces justified judgements without needing a proof.

8.2 Lemmas

This section gives several lemmas that will be used in the full proof of correctness of the following section.

The purpose of the Writing Variables Lemma for Types is to show that a type judgement remains correct after a variable is written.

Lemma 8.1 (Writing Variables for Types)**.** Let $cfg = (act, cnt) = \mathbf{step}_n(\mathcal{P})$, l be any label for a writable variable in cfg, *object* be a valid object for cfg, and $var :_{ctx} type$ be a type judgement that is correct for cfg. Suppose one of the following is true:

1. $var \neq \mathbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, l)$.

2. ctx does not match act.

3. obj is of type $type$.

Then $var :_{ctx} type$ is correct for **write_var**(cfg,l,obj).

Proof. Let $cfg' = \mathbf{write\_var}(cfg, l, obj)$. Note that $\mathbf{all\_activations}(cfg) = \mathbf{all\_activations}(cfg')$, i.e. there is no activation that is either created or destroyed by writing a variable. Consider each $act \in \mathbf{all\_activations}(cfg)$ in turn. The definition in subsection 6.7.1 requires the following property of *act*:

$$\begin{aligned} & ctx(act, cfg) \ \wedge \ var = \mathbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, var.label) \\ & \quad \Rightarrow \mathbf{read\_var}(cfg, act, var.label) \in type \end{aligned}$$

Let cid_{mod} be the contour that is modified by the call to **write_var**:

$$cid_{mod} = \mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act, l, \texttt{false})$$

Suppose that $ctx(act, cfg)$ and that $var = \mathbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, var.label)$, because otherwise the required property is vacuously true. There are several cases, all straightforward.

Suppose $l = var.label$ but that the contour that was written is different from the contour that l will be read from in *act*:

$$cid_{mod} \neq \mathbf{lookup\_contour}_{\mathcal{P}}(cfg, act, l, \texttt{false})$$

Then, **write_var** would not modify the contour l will be read from in *act*, and thus reading l from *act* will give the same object as before. Thus the required property will remain true.

Suppose that $var \neq \mathbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, l)$. If $l \neq var.label$ then the object read from cfg' is the same as from *cfg* and thus the required property is trivially true. Otherwise, by the Unshared Contours Lemma from subsection 6.2.4, the contour modified by the **write_var** must be different from that read in cfg', and thus the object read must still be the same in both cfg' and *cfg*.

Suppose that all of the above cases are false and that *ctx* does not match *cfg.act*. Then it is impossible for both *ctx* to match *act* and cid_{mod} to be the contour that l would be read from in *act*. Since the type judgement is well-formed, any parameter mentioned in *ctx* must be readable from any activation where *var* may be read. Since by assumption the contour *var* is read from is cid_{mod}, the objects read for each parameter mentioned in *ctx* must be the same in both *act* and *cfg.act*. Thus, *ctx* does not match *act* and the necessary property is vacuously true.

Finally, if all of the other cases are false, then $object \in type$ and the contour read from is cid_{mod}. In this case, the necessary property is trivially true. □

The following two lemmas are used to show that a flow judgement remains correct across one step of execution, with respect to one particular object. The first lemma concerns the case where a step of execution writes a different object than the object of interest. Informally, when the first lemma applies, it is said that "the flow of the object does not increase".

Lemma 8.2 (Writing Different Objects for Flow)**.** If $object \neq object'$, and l is a label for a writable variable of configuration *cfg*, and:

$$cfg' = \mathbf{write\_var}(cfg, l, object)$$

then:

$$\mathbf{flowpos}(object, cfg') \sqsubseteq \mathbf{flowpos}(object, cfg)$$

Proof. Note that the flow position of an object in a configuration may be computed by enumerating the activations and contours of the configuration, selecting the contours and activations that bind a variable or method receiver to the object, and then taking the union of the simple flow positions designating those bindings.

In this case, the configurations *cfg* and cfg' are the same except that one contour has one variable rebound. Let *cid* be the id of the contour that changes. If *cid* does not bind l to *object* in *cfg*, then same contours and activations will contribute the same simple flow positions to the overall flow position of *object* in both *cfg* and cfg'. Thus the union of those flow positions will be the same for *cfg* and cfg'. If, however, *cid* does bind l to *object*, then there will be one fewer contribution in cfg'. That is,

$$\mathbf{flowpos}(object, cfg) = (\mathbf{flowpos}(object, cfg') \sqcup f)$$

for some f. Therefore,

$$\textbf{flowpos}(object, cfg') \sqsubseteq \textbf{flowpos}(object, cfg)$$

□

The second lemma concerns the case where the object of interest is written in a step of execution using **write_var**. In this case, the flow position of the object does increase. The lemma is used to show that a set of flow judgements correctly allows for the increase of flow position that occurs.

Lemma 8.3 (Writing Variables for Flow)**.** Let $cfg = (act, cnt) = \textbf{step}_n(\mathcal{P})$, l be any label for a writable variable in cfg, $object$ be a valid object for cfg, $\mathcal{G}$ be a set of flow judgements, $f = \textbf{lhs}(\mathcal{G})$, and $f' = \textbf{rhs}(\mathcal{G})$. Suppose $\textbf{flowpos}(object, cfg) \sqsubseteq f$. Let

$$var = \textbf{dynlookup\_var}_{\mathcal{P}}(cfg, act, l)$$

and

$$ctx = \textbf{minctx}_{\mathcal{P}}(cfg)$$

and

$$fpos = [: V\ var :]_{ctx}$$

and

$$cfg' = \textbf{write\_var}(cfg, var.label, object)$$

If $fpos \sqsubseteq f'$, then $\textbf{flowpos}(object, cfg') \sqsubseteq (f \sqcup f')$.

Proof. Again, one contour will be different in cfg and cfg'. Call it cid. That contour will contribute one new flow position f_{cid} to the flow position of $object$, i.e.

$$\textbf{flowpos}(object, cfg') = \textbf{flowpos}(object, cfg) \sqcup f_{cid}$$

where f_{cid} is the new contribution. If one considers each way that var might be determined with **dynlookup_var**, one sees that the resulting contribution f_{cid} will be exactly $fpos$ as declared above. Thus:

$$\begin{aligned}
\textbf{flowpos}(object, cfg') &= \textbf{flowpos}(object, cfg) \sqcup fpos \\
\textbf{flowpos}(object, cfg) \sqcup fpos &\sqsubseteq f \sqcup f' \\
\textbf{flowpos}(object, cfg') &\sqsubseteq f \sqcup f'
\end{aligned}$$

□

The next lemma gives a basis for reasoning about where selector objects for a particular selector may be found as the program runs.

Lemma 8.4 (Flow Position of Selectors)**.** Suppose that:

- sel is any selector.
- For each literal statement:

  $$([\![l := sel]\!], b) \in \textbf{bound\_stats}(\mathcal{P})$$

  that instantiates selector sel, it is true that $\exists i : f_i = [: V\ b[l] :]_{\top_{ctx}}$.
- The judgements $f_1 \rightarrow^* f_1', \ldots, f_p \rightarrow^* f_p'$ are correct for configurations $\textbf{step}_0(\mathcal{P}) \ldots \textbf{step}_{n+1}(\mathcal{P})$.
- $fp = \bigsqcup_{i=1}^{p} f_i'$

Then, for any selector object $so \in \textbf{all\_objects}_{\mathcal{P}}(\textbf{step}_{n+1}(\mathcal{P}))$ whose selector is sel, both of the following are true:

- There is an $m \leq n+1$ such that the flow position of *so* in $\mathbf{step}_m(\mathcal{P})$ is subsumed by f_i for some i.

- *so* is within flow position *fp* in $\mathbf{step}_{n+1}(\mathcal{P})$.

Proof. The proof is by induction on the number of steps of execution.

The base case is trivial: there are no selector objects at all in $\mathbf{all\_objects}_{\mathcal{P}}(\mathbf{step}_0(\mathcal{P}))$.

Assume, then, that the lemma is correct for configurations $0 \ldots n$, and we will show that it must also be correct for configuration $n+1$.

Suppose the next statement to execute in $\mathbf{step}_n(\mathcal{P})$ is a literal statement $l := sel$, under static bindings b, that instantiates the selector of interest. The statement creates a new selector object whose flow position in $\mathbf{step}_{n+1}(\mathcal{P})$ is exactly $[\!: V\ b[l]\ :\!]_c$ for some context c. Since $([\![l := sel]\!], b) \in \mathbf{bound\_stats}(\mathcal{P})$, there must be an f_i which subsumes $[\!: V\ b[l]\ :\!]_c$, thus satisfying the first claim of the lemma for the newly created selector object. Regarding the second claim, notice that whenever $f \rightarrow^* f'$ holds true, it must be that $f \sqsubseteq f'$. Since $f_i \rightarrow^* f_i'$, the flow position of the new selector object must be subsumed by f_i' as well by f_i. Since $fp = \bigsqcup_{i=1}^{p} f_i'$, the object must also be in position *fp*.

Now consider any other selector object $so \in \mathbf{all\_objects}_{\mathcal{P}}(\mathbf{step}_{n+1}(\mathcal{P}))$ whose selector is *sel*. Inspection of the semantics will show that any such object must also be present in the previous configuration, i.e., $so \in \mathbf{all\_objects}_{\mathcal{P}}(\mathbf{step}_n(\mathcal{P}))$. Further, the flow position of *so* is unaffected by the execution of $l := sel$. Therefore, for any such object *so*, the inductive hypothesis leads immediately to the same claims holding true for configuration $n+1$.

Finally, suppose some statement executes other than a literal statement instantiating *sel*. In that case, no new selector objects for *sel* appear in $\mathbf{step}_{n+1}(\mathcal{P})$. The first claim of the lemma is satisfied by choosing the same i's for each selector object that, due to the inductive assumption, must have been available in $\mathbf{step}_n(\mathcal{P})$. The second claim is slightly less trivial, because the flow positions of the existing *sel* selector objects may have increased. For any such object *so*, however, the first claim that the flow position of *so* was subsumed by one of the f_i's in an earlier configuration, when combined with the lemma's assumption that $f_i \rightarrow^* f_i'$ is true in configurations $0 \ldots n+1$, leads to the second claim. The flow position of *so* must be subsumed by f_i' and thus also subsumed by *fp*. □

The next lemma is similar to the last, except that it reasons about the location of block objects instead of selector objects.

Lemma 8.5 (Flow Position of Blocks)**.** Suppose that:

- *bs* specifies any block in $\mathcal{P}$.

- If

$$([\![l := block]\!], b) \in \mathbf{bound\_stats}(\mathcal{P})$$

  is the statement corresponding to *bs*, then $[\!: V\ b[l]\ :\!]_{\top_{ctx}} \rightarrow^* fp$ is true for configurations $\mathbf{step}_0(\mathcal{P}), \ldots, \mathbf{step}_{n+1}(\mathcal{P})$.

Then, for any closure $bobj \in \mathbf{all\_objects}_{\mathcal{P}}(\mathbf{step}_{n+1}(\mathcal{P}))$ whose block is *bs*, *bobj* is in the flow position *fp* in $\mathbf{step}_{n+1}(\mathcal{P})$.

Proof. The proof closely parallels that for the Flow Position of Selectors Lemma. □

8.3 Main theorem

The proof is subdivided into proofs of the following propositions for arbitrary n:

$$\begin{aligned} & F^*(0) \\ & T(0) \\ T(n) &\Rightarrow R(n) \\ (n = 0 \lor S(n-1)) \land T(n) \land F^*(n) &\Rightarrow S(n) \\ T(n) \land R(n) \land S(n) &\Rightarrow T(n+1) \\ R(n) \land (n = 0 \lor S(n-1)) &\Rightarrow F(n) \\ F^*(n) \land F(n) &\Rightarrow F^*(n+1) \end{aligned}$$

where:

- $T(n)$ means that the type justifications in $\mathcal{J}$ are correct for configuration $\mathbf{step}_n(\mathcal{P})$.
- $F(n)$ means that the flow justifications in $\mathcal{J}$ are correct for configuration $\mathbf{step}_n(\mathcal{P})$.
- $F^*(n)$ means that the transitive flow justifications in $\mathcal{J}$ are correct for the configurations $\mathbf{step}_0(\mathcal{P})$ through $\mathbf{step}_n(\mathcal{P})$.
- $R(n)$ means that the responders justifications in $\mathcal{J}$ are correct for configuration $\mathbf{step}_n(\mathcal{P})$.
- $S(n)$ means that the senders justifications in $\mathcal{J}$ are correct for configurations $\mathbf{step}_0(\mathcal{P})$ through $\mathbf{step}_n(\mathcal{P})$.

Given these propositions, it is then straightforward to show by induction that:

$$\forall n : T(n) \land F(n) \land F^*(n) \land R(n) \land S(n)$$

which is the desired theorem.

Each subsection below proves one of the above propositions to be true. In general, each section assumes that neither $\mathbf{step}_n(\mathcal{P})$ nor $\mathbf{step}_{n+1}(\mathcal{P})$ is halted, because otherwise the proof in that subsection is trivial.

8.3.1 Transitive flow judgements in the initial configuration

It is to be shown that:

$$F^*(0)$$

That is, it is to be shown that the transitive flow judgements of $\mathcal{J}$ are correct in the initial configuration.

For all $f \to^* f' \in \mathcal{J}$, it must be shown that $f \sqsubseteq f'$. Each such judgement must have been justified with either J-PRUNE-FTRANS or F-TRANS. If rule J-PRUNE-FTRANS was used, then $f' = \top_{fp}$, and thus it must be that $f \sqsubseteq f'$. If F-TRANS was used, then $f \sqsubseteq f'$ is directly listed as an assumption of the justification.

8.3.2 Type judgements in the initial configuration

It is to be shown that:

$$T(0)$$

That is, it is to be shown that the type judgements of $\mathcal{J}$ are correct in the initial configuration.

There are no parameters bound in the initial configuration, and thus any type judgements regarding parameters are trivially true. The type judgements involving non-parameter variables must be justified with either JUST-PRUNE-TYPE or JUST-ONE; they cannot be justified with JUST-CTX, because only parameters have a type specified by a context. For any judgement that is justified by JUST-PRUNE-TYPE, the type is $\top$ and thus the judgement is trivially true. For any non-parameter type judgement justified by JUST-ONE, the type must subsume $[\![\text{UndefinedObject}]\!]$. Since every variable binding in the initial configuration binds to `NilObj`, these judgements are correct in the initial configuration.

8.3.3 Responders judgements

It is to be shown that:

$$T(n) \Rightarrow R(n)$$

That is, it is assumed that the type judgements are correct in configurations up to n, and it must be shown that these assumptions imply that the responders judgements are true in configuration n.

A responders judgement $stat_{ctx} \star b \xrightarrow{send} rs$ must be justified by one of the rules R-SEND, R-SENDVAR, or R-BEVAL, JUST-PRUNE-RESP, depending on the form of $stat$ and on whether rs is non-trivial. To avoid triviality, suppose that $rs \neq \top_r$, that $stat$ is about to execute, that $b[\texttt{block}]$ is the block of the main activation, and that ctx matches the main activation of cfg. Under these assumptions, JUST-PRUNE-RESP could not have been used.

Consider `send` statements first. That is, suppose:

$$stat = [\![l := \texttt{send}(l_{rcvr}, selector, l_1 \ldots l_m)]\!]$$

The rule R-SEND must have been used. By the Lexical Binding Lemma, the variables v_{rcvr}, etc., deduced from l_{rcvr}, etc., by using the binding map b, are the same as would be found with **dynamic_bindings** on the main activation of cfg. By the inductive assumption, and by the proofs above for correctness of type judgements, all of the type judgements required by the assumption of R-SEND are correct for cfg. Thus, the object found by reading l_{rcvr} is a member of type t_{rcvr}, and likewise the object found by reading any l_i is a member of type t_i.

Thus, **lookup*** will have correct information and so the method about to be invoked is one of the methods $m_1 \ldots m_n$. Suppose the method is m_j. Further, the context of the new activation must match c_j. Since **cpasplit** has full inclusiveness of interpretations (Theorem 3), it must further be that the new activation matches one of the split contexts $c_{(j,k)}$. By the last assumption of R-SEND, it must be that

$$(m_j, c_{(j,k)}) \in rs$$

Thus the judgement is correct.

The proofs for `sendvar` and `beval` statements are close parallels. Note that the justification rules implicitly require, respectively, that there are a finite number of possible selectors possibly held by $selvar$ or a finite number of blocks possibly held by $blockvar$. If this criterion fails, then these justification rules cannot have been used and thus the responder set must be $\top_r$.

8.3.4 Senders judgements

It is to be shown that:

$$(n = 0 \vee S(n-1)) \wedge T(n) \wedge F^*(n) \Rightarrow S(n)$$

That is, it is assumed that the type judgements and transitive flow judgements are correct in configuration in $\mathbf{step}_n(\mathcal{P})$ and that the senders judgements are correct in configurations up to the previous configuration, and it must be shown that the senders judgements are also true in configuration n. Let $cfg = \mathbf{step}_n(\mathcal{P})$ and $cfg' = \mathbf{step}(cfg)$.

Consider any senders judgement $blk_{ctx} \xleftarrow{send} ss$. To avoid triviality, suppose that $ss \neq \top_s$. If $stat$ is the statement about to execute in cfg, and b is the binding map of the main activation of cfg, then there must be a justification that:

$$stat \star b \quad \triangleright \quad blk_{ctx} \xleftarrow{send} ss$$

Suppose the justification is one of S-SELF, S-LIT, S-SEL, S-VAR, S-NEW, or S-BLOCK. Then, $stat$ is not a message-sending statement and the judgement is true. If the justification is instead one of S-SEND-TRIV or S-SENDVAR-TRIV, then the main activation of cfg' cannot be an activation for blk; blk

is not the block for a method. Similarly, if the justification is S-BEVAL-TRIV, then the main activation will not be for a method, whereas *blk* is for a method.

If the justification is S-SEND-BADSELECTOR, then the main activation of *cfg′* must be for a method other than *blk*'s method; the selectors do not match.

Suppose then that the justification is S-SEND-BADRECV. Since, by assumption, type judgements are correct, the receiver object for the `send` must be a member of type t_{rcvr}. By assumption, the method of *blk* is not a method that may be invoked by any member of t_{rcvr}, and thus the method of *cfg′* must not be the method of *blk*.

If the justification is S-SEND, then the statement and context are in *ss* and thus the judgement is correct.

Suppose the justification is S-SENDVAR. Then, by the Flow Position of Selectors Lemma, the only selector objects in **all_objects**(*cfg*) that match the selector of *bs*'s method are within the flow position f_{sel} as calculated in the assumptions of S-SENDVAR. If f_{sel} includes no variable flow position matching *selvar* and the current activation, then the selector sent by this `sendvar` execution cannot match *bs*'s method. Further, just as with S-SEND, if t_r does not include any objects such that the `sendvar` could invoke *bs*'s method, then this `sendvar` execution cannot invoke *bs*'s method. Under all other circumstances, there will be a tuple in *ss* that matches the current activation of *cfg*. In any of these cases, the senders judgement is still correct.

If the justification is S-BEVAL, then the proof parallels that for S-SENDVAR, only using the Flow Position of Blocks Lemma instead of the Flow Position of Selectors Lemma.

8.3.5 Type judgements

It is to be shown that:

$$T(n) \wedge R(n) \wedge S(n) \Rightarrow T(n+1)$$

That is, it is assumed that the type judgements and responders judgements of $\mathcal{J}$ are true for configuration n, and it is also assumed that the senders judgements of $\mathcal{J}$ are true in configurations $0 \ldots n$. It must be shown that these assumptions are sufficient to imply that the type judgements remain true in configuration $n+1$.

Let $cfg = \mathbf{step}_n(\mathcal{P})$ and $cfg' = \mathbf{step}_{n+1}(\mathcal{P})$. Consider any type judgement $var :_{ctx} type \in \mathcal{J}$. By assumption, this judgement must be justified by some justification rule.

If the judgement is justified by JUST-PRUNE-TYPE, then $type = \top$. Since all objects are members of this type, the judgement is correct.

If JUST-CTX rule is used to justify the judgement, then the result is true tautologically. If a context matches an activation in a configuration, then any variable read in that activation must yield an object in the type specified by the activation.

The only other possible justification is JUST-ONE. The rest of this section assumes that the judgement is justified with JUST-ONE, and thus that there is a justification that the judgement accounts for each statement in the program.

Suppose that *var* is a parameter. If the statement about to execute in *cfg* is not a `send`, `beval`, or `sendvar` statement, or if the current block is about to return a value, then the set of objects bound to *var* in *cfg′* will be a non-strict subset of the objects bound to *var* in *cfg*. To see this, observe that **write_var** never modifies a parameter binding, and thus the only way to bind a new object to *var* is to create a new activation. In none of these cases is a new activation created.

Three cases remain if *var* is a parameter. Suppose first that the statement about to execute is:

$$l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)$$

This statement can only be accounted for by T-SEND, which in turn requires that one of the rules T-SEND-S and T-SEND-S-TRIV is used to justify that the judgement accounts for the execution of this `send` statement. T-SEND-S-TRIV cannot actually be used in this case, however, because it requires *var* not be a parameter. Thus T-SEND-S must have been used. T-SEND-S requires that a senders judgement has been justified for *var*'s method, and by assumption this judgement must be correct for *cfg*. Thus if *var*'s method is the main method of *cfg′*, there must be a tuple $(stat, ctx)$ among the senders that have been found, where *stat* is the statement about to execute and *ctx* matches the main activation of *cfg*. T-SEND-S thus requires that there has been a type judgement for the relevant argument of the `send` statement. By assumption, that type judgement is correct in *cfg*. Since T-SEND-S requires that *type* subsume this type, and since by assumption *type* subsumes the type of *var* in *cfg*, *type* must also subsume the type of *var* in *cfg′*.

The final two cases where *var* is a parameter are those where the statement about to execute is a `beval` or `sendvar` statement:

$$l_l := \texttt{beval}(l_b, l_1 \ldots l_m)$$
$$l_l := \texttt{sendvar}(l_{rcvr}, selvar, l_1 \ldots l_m)$$

In both cases, the reasoning exactly parallels that for `send` statements.

Now suppose that *var* is not a parameter.

Suppose the *statement* about to execute is:

$$l := \texttt{self}$$

The semantics of this statement are that the current receiver is written into variable *l*.

There must be some justification that the judgement accounts for this statement. If the justification is by T-SELF-TRIV, then $var \neq b[l]$; since Mini-Smalltalk is statically bound, then also $var \neq$ **dynlookup_var**$_{\mathcal{P}}(cfg, act, l)$. Thus, by the Writing Variables Lemma, the judgement remains true in *cfg′*.

If, however, $var = b[l]$, then the justification must be by T-SELF. Suppose that *ctx* matches the current configuration; otherwise the Writing Variables Lemma is enough to prove that the judgement remains true in *cfg′*. By the assumptions of T-SELF, *type* must be a supertype of the intersection of two types: a cone type for the class the method belongs to, and *ctx*[`self`]. By the Semantic Sanity Lemma, the current receiver must in fact be an element of the cone type. Further, by assumption, the current receiver is an element of type *ctx*[`self`]. Since the receiver matches both types, it also matches the intersection of those types, as systematic inspection of the definition of meet for types (Figure 6.6) and contexts (Figure 6.9) will verify. Thus, again by the Writing Variables Lemma for Types, the type judgement remains correct in configuration $\textbf{step}_{n+1}(\mathcal{P})$.

Next, suppose that the statement about to execute is:

$$l := l'$$

If *var* is not *l*, the variable being written, or if *ctx* does not match *cfg*, then the Writing Variables Lemma implies that the type judgement remains true. These two trivial cases appear for every statement type below, and since the proof is the same, it will not be repeated.

The remaining case is that *var* is *l*, the variable that is written. There must have been a T-VAR justification used to account for this statement, in which case, *type* will be a supertype of some *type′* where $l' :_{\lceil ctx \rceil} type' \in \mathcal{J}$. Since the type judgements of $\mathcal{J}$ are assumed to be true for *cfg*, it must be that the object read from *l′* is in *type′*. Thus the object is also in *type*, and the Writing Variables Lemma implies that the judgement remains correct.

Next, suppose that the statement about to execute is:

$$l := \texttt{new}\ \ cname$$

The object written will be of class *cname*. If *var* is bound to *l*, then T-NEW will have been used, and *var* will include the class type of *cname* and thus will include the newly created object.

Next, suppose that the statement about to execute is:

$$l := block$$

A new closure is created. If *var* is written, then the type judgement will have been justified by T-BLOCK, and *type* will include the type of the closure.

Next, suppose that a send statement is about to execute:

$$l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)$$

or:

$$l_l := \texttt{beval}(l_b, l_1 \ldots l_m)$$

or:

$$l_l := \texttt{send}(l_{rcvr}, selvar, l_1 \ldots l_m)$$

In all three cases, only parameters are modified, and so these cases are trivial.

Next, suppose that the current block is ending. Suppose that its *retFromMethod* is true and that the block returns l. By the Send History Lemma, either execution halts, or the variable written was on the left hand side of a `send` or `sendvar` statement of a previous execution. If *var* is the variable that is written, then the current method must have been one of the m_i's considered in the T-SEND-R or T-SENDVAR-R justification for this statement. Furthermore, the current activation must be matched by one of the contexts $c_{(i,j)}$. Thus, the type returned must have been correctly predicted, and that type will be included in the type of *var*.

Finally, suppose that the current block is ending, that the block's *retFromMethod* is false, and that the block returns l. The proof is similar to that for method returns. The `beval` statement is correctly located, the context of the current activation was one of those that was predicted, and the type of the left hand side of the `beval` statement will include the type that is returned.

All cases have now been considered. The type judgements of $\mathcal{J}$ remain correct across an invocation of **step**.

8.3.6 Simple flow judgements

It is to be shown that:

$$R(n) \wedge (n = 0 \vee S(n-1)) \Rightarrow F(n)$$

That is, it is assumed that the responders judgements of $\mathcal{J}$ are true for configuration n, and that, unless $n = 0$, the senders judgements are true in configuration $n-1$. It must be shown that the simple flow judgements of $\mathcal{J}$ are true in configuration n.

As usual, let $cfg = (act, cnt) = \mathbf{step}_n(\mathcal{P})$, and $cfg' = \mathbf{step}(cfg)$. Consider any non-`nil` object *object* with a non-empty flow position in *cfg*. Also consider any subset $\mathcal{G}$ of the flow judgements in $\mathcal{J}$ such that $\mathbf{lhs}(\mathcal{G})$ subsumes the flow position of *object* in *cfg*. It will be shown that the flow position of *object* in cfg' will be subsumed by $\mathbf{lhs}(\mathcal{G}) \sqcup \mathbf{rhs}(\mathcal{G})$. Since the argument holds for any $\mathcal{G}$ and any *object*, the set of simple flow judgements in $\mathcal{J}$ must be correct for *cfg*.

First, suppose the statement about to execute is:

$$l := \texttt{self}$$

If $act.rcvr \neq object$, then the Writing Different Objects Lemma implies that the flow position of *object* in cfg' will be subsumed by its flow position in *cfg*, and thus the desired property is true. Therefore, suppose that the current receiver is *object*, i.e. $act.rcvr = object$.

Since by assumption $\mathbf{lhs}(\mathcal{G})$ subsumes the flow position of *object*, and since the left-hand side of a justified flow judgement cannot be $\top_{fp}$ or a sum flow position, there must be a flow judgement in $\mathcal{G}$ whose left-hand side is a self flow position for the current method and whose context matches the current activation:

$$\begin{array}{c}[\!:\ S\ \ cfg.act.block.meth\ :\!]_{ctx} \rightarrow f' \quad \in \quad \mathcal{G} \\ where\ \ ctx(act, cfg)\end{array}$$

Since this judgement has been justified, either $f' = \top_{fp}$, in which case the result is trivial, or the judgement must be justified by JUST-ONE. Suppose it is justified by JUST-ONE. It follows from the Lexical Binding Lemma that there must be a tuple $([\![l := \texttt{self}]\!], b) \in \mathbf{bound\_stats}(\mathcal{P})$, where b matches the static variables visible from the main activation of *cfg*. Thus, by the assumptions of JUST-ONE, there must be a justification of:

$$[l := \texttt{self}] \star b \quad \rhd \quad [\colon S\ \mathit{cfg.act.block.meth}\ \colon]_{ctx} \rightarrow f'$$

Only F-SELF may be used to justify this assertion. Thus, f' must be a variable flow position for $b[l]$:

$$[\colon V\ b[l]\ \colon]_{\lceil ctx \rceil} \sqsubseteq f'$$

By the Lexical Binding Lemma, $b[l]$ must be the same as the static variable found dynamically by starting at *cfg*. Thus, by the Writing Variables Lemma for Flow, the flow position of *object* in *cfg′* is subsumed by $\mathbf{lhs}(\mathcal{G}) \sqcup \mathbf{rhs}(\mathcal{G})$.

Next, suppose that the statement about to execute is:

$$l := \mathit{literal}$$

The semantics instantiate a new object for the literal. Since the object is required to have a new contour, it must be different from *object*, and thus the Writing Different Objects Lemma applies. The flow of *object* does not increase after a literal statement.

Next, suppose that the statement about to execute is:

$$l := l'$$

Suppose the flow position of *object* includes a variable flow position for l', in the minimum context of *cfg*. (Otherwise, the flow position of *object* does not increase.) The subset of flow judgements must include one judgement $f \rightarrow f'$ where f is a variable flow position for l'. Unless $f' = \top_{fp}$ (a trivial case), the judgement must have been justified with F-VAR. Thus f' must include a variable flow position for l in a context matching *cfg*. Thus, $\mathbf{lhs}(\mathcal{G}) \sqcup \mathbf{rhs}(\mathcal{G})$ must include the flow position of *object* in *cfg′*.

Next, suppose that the statement about to execute is:

$$l := \texttt{new}\ \ l'$$

The flow position of *object* remains the same. The only flow position that increases is that of the newly created object, and *object* cannot be that object because *object* had a non-empty flow position in *cfg*.

Next, suppose that the statement about to execute is:

$$l := \mathit{block}$$

Again, the flow positions of existing objects do not change from *cfg* to *cfg′*.

Next, suppose that the statement about to execute is:

$$l_l := \texttt{send}(l_{rcvr}, sel, l_1 \ldots l_m)$$

Consider each label among l_{rcvr} and $l_1 \ldots l_m$. For each of these that is bound to *object* in *cfg*, there must be a judgement $f_l \rightarrow f_l'$ in $\mathcal{G}$ where f_l is a variable flow position for the variable the label binds to in *cfg*. These judgements must account for the statement about to execute, and that accounting must be justified by either F-SEND-SELF or F-SEND-PARAM, both of which have the same structure. Both F-SEND-SELF and F-SEND-PARAM require that there be a senders judgement on the statement about to execute. By assumption, each such senders judgement is correct in *cfg*, and thus the method about to execute must be among those predicted by the senders judgement. Each of these rules also requires that a flow position corresponding to the receiver or appropriate parameter of the responding method is included in the right hand side of the flow judgement. Thus, for each new position in *cfg′* that binds *object*, there is a flow judgement in $\mathcal{G}$ whose right hand side includes that position. Thus, all new bindings of *object* in *cfg′* are included in $\mathbf{rhs}(\mathcal{G})$, and thus the necessary criterion for $\mathcal{G}$ is met in *cfg*.

Next, suppose the statement about to execute is:

$$l_l := \texttt{sendvar}(l_{rcvr}, selvar, l_1 \ldots l_m)$$

or:

$$l_l := \texttt{beval}(l_b, l_1 \ldots l_m)$$

The reasoning exactly parallels that for `send` statements, except that for `beval` statements, there is no flow into `self` positions.

Next, suppose that the block is ending. Suppose, to avoid triviality, that the variable var_{ret} that is being returned from the block is bound to *object*. Thus, there must be some judgement $f \to f' \in \mathcal{G}$ where f is a variable flow position for var_{ret} in a context matching *act*. By the Send History Lemma, there is a statement that, in a previous step of execution, invoked the method that is currently returning. There must be an invocation of F-RETURN-SEND, F-RETURN-SENDVAR, or F-BRETURN-BEVAL to allow $f \to f'$ to account for returns to this statement from the current method of *cfg*. It is impossible that one of the trivial rules, F-RETURN-SEND-BADVAR, F-RETURN-SENDVAR-BADVAR, or F-BRETURN-BEVAL-BADVAR, was used, because var_{ret} is in fact among the returned variables of the current method.

All three of the non-trivial F-RETURN rules have the same pattern. They require that there is a senders judgement in $\mathcal{J}$ for the returning method. By assumption, senders judgements are correct in *cfg*, and thus the senders judgement required by the rule must include the statement where control is returning. Thus, the rule requires f' to hold a variable flow position for the variable on the left hand side of the statement where control is returning. Thus, f' includes the new binding of *object*, and the necessary criterion is met.

8.3.7 Transitive flow judgements

It is to be shown that:

$$F(n)^* \wedge F(n) \Rightarrow F^*(n+1)$$

The assumption is that both the simple and transitive flow judgements of $\mathcal{J}$ are correct for steps $0 \ldots n$. It is to be shown that the transitive flow judgements of $\mathcal{J}$ are true for steps $0 \ldots n+1$.

Refer to the definition of correct sets of transitive flow goals in subsection 6.7.3. Let $\mathcal{F}$ be the set of transitive flow judgements in $\mathcal{J}$, and let $\mathcal{G}$ be any subset of $\mathcal{F}$. Let *object* be any object in configuration $i \leq n$ other than `NilObj` with a non-empty flow position in $\mathbf{step}_i(\mathcal{P})$. To avoid triviality, suppose that:

$$\mathbf{flowpos}(object, \mathbf{step}_i(\mathcal{P})) \sqsubseteq \mathbf{lhs}(\mathcal{G})$$

By the inductive assumption, it is known that for any $j \in i \ldots n$

$$\mathbf{flowpos}(object, \mathbf{step}_j(\mathcal{P})) \sqsubseteq \mathbf{rhs}(\mathcal{G})$$

It must be shown that:

$$\mathbf{flowpos}(object, \mathbf{step}_{n+1}(\mathcal{P})) \sqsubseteq \mathbf{rhs}(\mathcal{G})$$

Each transitive flow judgement in $\mathcal{G}$ must be justified, either by rule F-TRANS or by rule JUST-PRUNE-TFLOW. If any of them are justified by JUST-PRUNE-TFLOW, then the target of that judgement must be $\top_{fp}$, and $\mathbf{rhs}(\mathcal{G})$ must also be $\top_{fp}$. In that case, the criterion is trivially satisfied. Thus, suppose that none of the goals are justified by JUST-PRUNE-TFLOW, and therefore that all of them are justified by F-TRANS.

To meet the requirements of F-TRANS, for each judgement $f \to^* f' \in \mathcal{G}$, there must be a decomposition of f' into simple flow positions $f'_1 \ldots f'_p$, where, for each f'_i, there must be a simple flow judgement $f'_i \to f''_i \in \mathcal{J}$. Let $\mathcal{H}$ be the set containing all of these required $f'_i \to f''_i$ judgements but no others.

The flow position of *object* in $\mathbf{step}_n(\mathcal{P})$ is subsumed by $\mathbf{lhs}(\mathcal{H})$. Since it was just proven that the simple flow judgements are correct for $\mathbf{step}_n(\mathcal{P})$, and since $\mathcal{H}$ is a subset of the flow judgements of $\mathcal{J}$, it must be that the flow position of *object* in $\mathbf{step}_{n+1}(\mathcal{P})$ is also subsumed by $\mathbf{lhs}(\mathcal{H}) \sqcup \mathbf{rhs}(\mathcal{H})$. By the construction of $\mathcal{H}$, it must be that $\mathbf{lhs}(\mathcal{H}) \sqsubseteq \mathbf{rhs}(\mathcal{G})$. Since $\mathcal{G}$ holds only justified flow judgements, it must also be that $\mathbf{rhs}(\mathcal{H}) \sqsubseteq \mathbf{rhs}(\mathcal{G})$. Thus, the flow position of *object* in $\mathbf{step}_{n+1}(\mathcal{P})$ is also subsumed by $\mathbf{rhs}(\mathcal{G})$.

Therefore, the correctness requirement is satisfied for $\mathcal{G}$ and *object*. Since $\mathcal{G}$ and *object* are arbitrary, the set of all transitive flow judgements in $\mathcal{J}$ must also be correct.

Chapter 9

Chuck: Semantic program navigation

Chuck is a new program understanding tool for Squeak, a dialect of Smalltalk. It is both an example application of **DDP**, and an exhibition of the authors' motivation for develpoing **DDP**. Concretely, it extends the Refactoring Browser [56] with new queries and windows for data-flow information.

Chuck is available on the World Wide Web and is a standard load option of Squeak.

9.1 Semantic navigation

Chuck exploits the demand-driven structure of **DDP** to achieve a thorough integration of data-flow analysis into the existing development style supported by Squeak. Squeak's code browser already includes *context menus* which allow a user to highlight an item of interest in the code and then perform queries on those items. Chuck uses the same context menus but adds additional queries to them that are answerable now that a data-flow analysis is available.

The response to a query is not only an answer is always given in a *derivation browser*. The derivation browser shows the justification that **DDP** found for its answer. That justification is a trace through the code along *semantic* navigation paths. That is, the existing tools includes *syntactic* navigation paths such as the answer to "find references to this variable." Chuck maintains the same style of interaction for semantic navigation paths such as "find the invokers of this method."

9.2 Available queries

Chuck implements two new queries and two enhanced versions of standard Squeak queries. The two enhanced queries are used to trace call graphs statically. They are type-sensitive versions of the *implementers-of* and *senders-of* queries. The standard implementors-of query lets a programmer find all methods whose name matches a selection, whereas the standard senders-of query locates all message-send expressions that send the specified message.

The enhanced implementors-of query finds only those methods that, based on DDP type analysis, might actually respond to the selected message-send expression. As an extreme example, if a user browses to class `BasicLintRuleTest`'s `new` method in Squeak 3.7 and selects the message send of `initialize`, the standard query shows 756 potential responders. The enhanced query shows only one. Another example can be seen by comparing Figure 9.1 and Figure 9.2.

Similarly, the enhanced senders-of query returns only those message-send expressions that, based on type information, may invoke a specified method. To repeat the previous example in reverse, the standard tool shows 581 possible senders of `BasicLintRuleTest`'s `initialize` method, while Chuck shows 13. Another example can be seen by comparing Figure 9.3 and Figure 9.4.

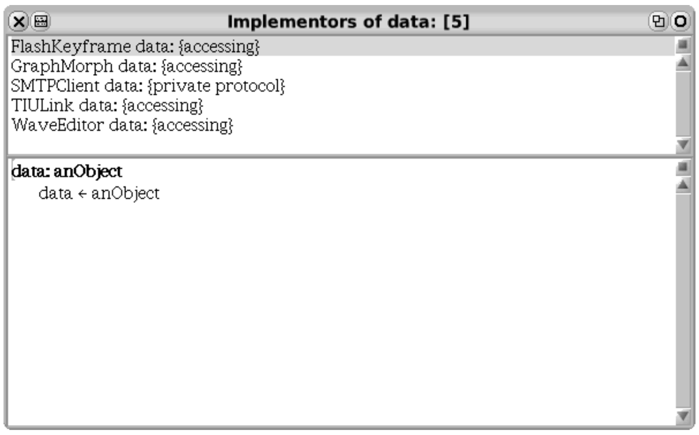
The two new queries are used to trace data flow: *type queries* and *forward-flow queries*. A type query lets the user select an expression or variable and then find, based on analysis information, what types the expression or variable may hold at runtime. Figure 9.5 shows the programmer asking for a type, and Figure 9.6 shows how Chuck displays the answer. Similarly, a forward-flow query lets the user find the expressions or variables in the program that values may reach, if they start at the selected expression or variable. Such a query is useful, for example, to see where a constant in the code is ultimately used. Figure 9.7 shows the programmer asking for the flow of an expression, and Figure 9.8 displays Chuck's answer.

9.3 Browsing derivations and trying harder

Chuck not only returns a judgement in response to a query, but can also return the support for a judgement. One may point to any judgement returned, and find out the other judgements the analyzer used to reach that conclusion. One may then recursively query *those* judgements to see how they, in turn, are justified. Figure 9.6 demonstrates this functionality.

Chuck's derivation browser provides two additional navigation links beyond the explanations. Support for a judgement usually involves reference to other elements of the source code. By selecting a judgement, the programmer can cause the code browser to jump to the relevant source code in the standard code browser. Again, Figure 9.6 demonstrates this functionality; the user is about to view the code underlying one of the judgements Chuck has produced.

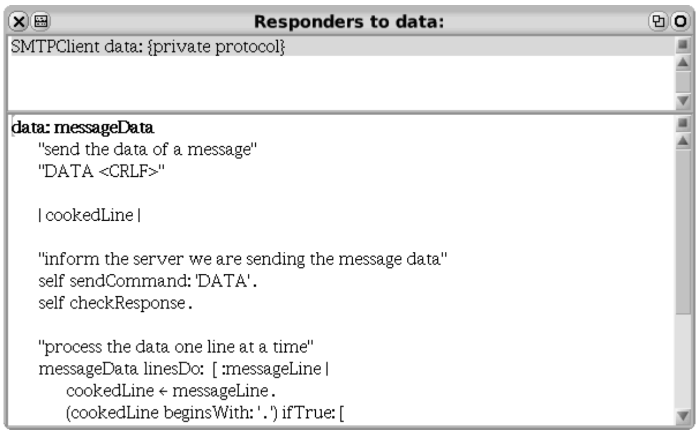
Second, the user may select any judgement and ask the analyzer to try harder on that particular goal, i.e. to use a higher pruning threshold. This ability lets Chuck give fast, imprecise answers by default, yet still allow users to allocate more time for a question if the question is important enough, and the first answer imprecise enough, to warrant the extra resource expenditure. The user in Figure 9.9 is requesting that the main goal be tried again. In Figure 9.10, the user specifies that greater resources should be used. The result is Figure 9.11, which is a precise result thanks to the increased resources.



```
FlashKeyframe data: {accessing}
GraphMorph data: {accessing}
SMTPClient data: {private protocol}
TIULink data: {accessing}
WaveEditor data: {accessing}
```

```
data: anObject
    data ← anObject
```

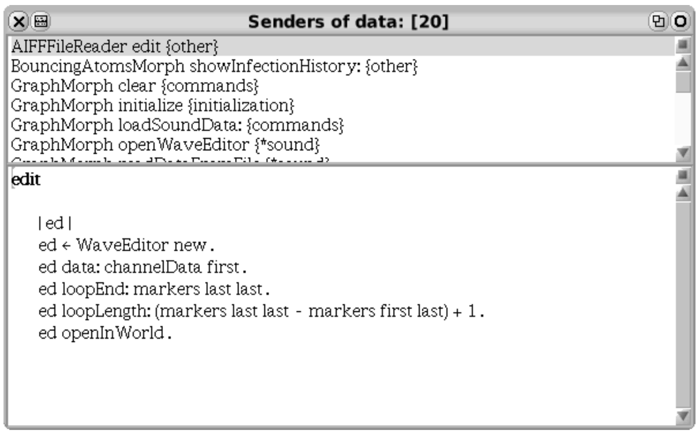
Figure 9.1: The standard tools show the methods that potentially respond to a message-send statement.



```
SMTPClient data: {private protocol}
```

```
data: messageData
    "send the data of a message"
    "DATA <CRLF>"

    | cookedLine |

    "inform the server we are sending the message data"
    self sendCommand: 'DATA'.
    self checkResponse.

    "process the data one line at a time"
    messageData linesDo: [ :messageLine |
        cookedLine ← messageLine.
        (cookedLine beginsWith: '.') ifTrue: [
```

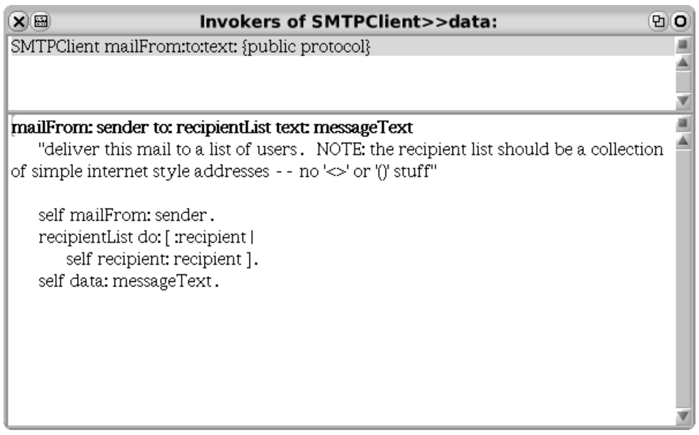
Figure 9.2: Chuck only displays potential responding methods that are consistent with its type inferences.



```
AIFFFileReader edit {other}
BouncingAtomsMorph showInfectionHistory: {other}
GraphMorph clear {commands}
GraphMorph initialize {initialization}
GraphMorph loadSoundData: {commands}
GraphMorph openWaveEditor {*sound}
```

```
edit

    | ed |
    ed ← WaveEditor new.
    ed data: channelData first.
    ed loopEnd: markers last last.
    ed loopLength: (markers last last - markers first last) + 1.
    ed openInWorld.
```

Figure 9.3: The standard tools show the statements that potentially invoke a method.



```
SMTPClient mailFrom:to:text: {public protocol}
```

```
mailFrom: sender to: recipientList text: messageText
    "deliver this mail to a list of users.  NOTE: the recipient list should be a collection
of simple internet style addresses -- no '<>' or '()' stuff"

    self mailFrom: sender.
    recipientList do: [ :recipient |
        self recipient: recipient ].
    self data: messageText.
```

Figure 9.4: Chuck only displays potential senders that are consistent with its type inferences.

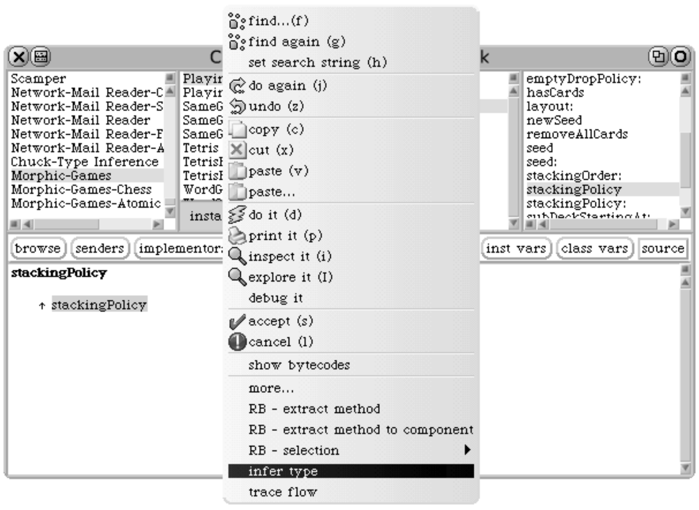
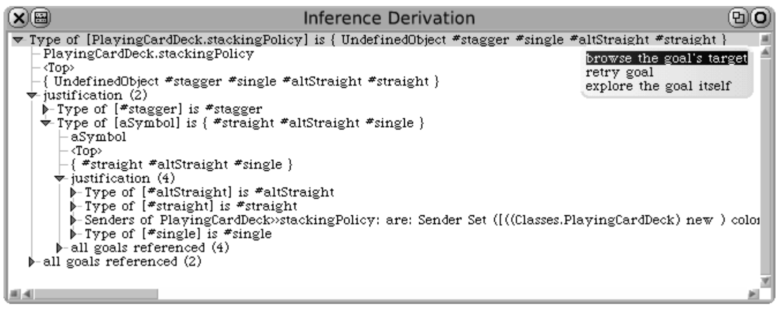
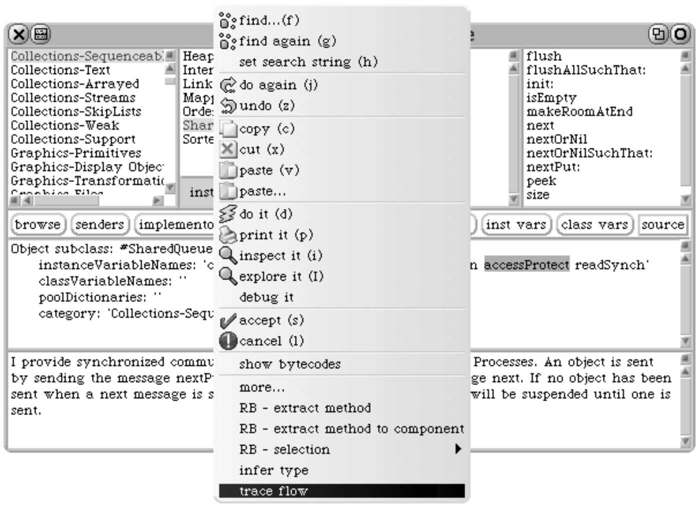


Figure 9.5: A user asks for the type of a variable.



```
Inference Derivation
▼ Type of [PlayingCardDeck.stackingPolicy] is { UndefinedObject #stagger #single #altStraight #straight }
    PlayingCardDeck.stackingPolicy
    <Top>
    { UndefinedObject #stagger #single #altStraight #straight }
  ▼ justification (2)
    ▶ Type of [#stagger] is #stagger
    ▼ Type of [aSymbol] is { #straight #altStraight #single }
        aSymbol
        <Top>
        { #straight #altStraight #single }
      ▼ justification (4)
        ▶ Type of [#altStraight] is #altStraight
        ▶ Type of [#straight] is #straight
        ▶ Senders of PlayingCardDeck>>stackingPolicy: are: Sender Set ([((Classes.PlayingCardDeck) new ) colo
        ▶ Type of [#single] is #single
      ▶ all goals referenced (4)
  ▶ all goals referenced (2)
```

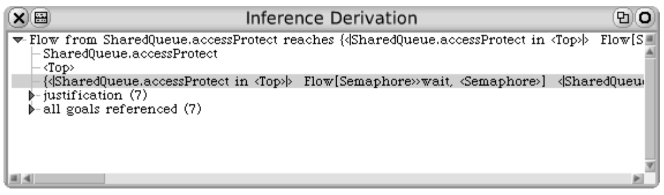
Figure 9.6: Chuck displays the type of a variable.



```
Object subclass: #SharedQueue
    instanceVariableNames: 'c                               n accessProtect readSynch'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Collections-Sequ
```

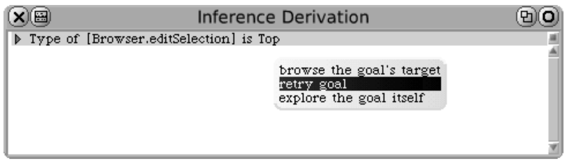
```
I provide synchronized commu                               Processes. An object is sent
by sending the message nextP                               ge next. If no object has been
sent when a next message is s                              will be suspended until one is
sent.
```

Figure 9.7: A user asks where a variable's contents flow.



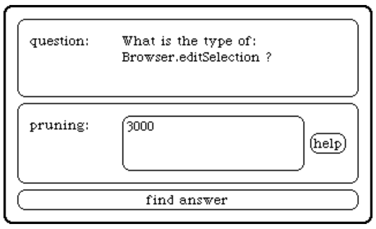
```
▼ Flow from SharedQueue.accessProtect reaches {<|SharedQueue.accessProtect in <Top>|>  Flow[S
    SharedQueue.accessProtect
    <Top>
    {<|SharedQueue.accessProtect in <Top>|>  Flow[Semaphore>>wait, <Semaphore>]  <|SharedQueu
  ▶ justification (7)
  ▶ all goals referenced (7)
```

Figure 9.8: Chuck displays the locations where a variable's contents flow.



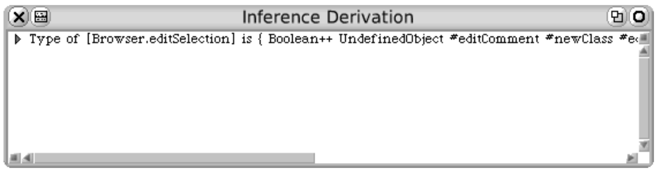
```
▶ Type of [Browser.editSelection] is Top
```

Figure 9.9: Sometimes Chuck fails due to lack of resources.



question:	What is the type of: Browser.editSelection ?
pruning:	3000

Figure 9.10: The user may "retry goal" and specify that more resources should be used the next time.



```
▶ Type of [Browser.editSelection] is { Boolean++ UndefinedObject #editComment #newClass #e
```

Figure 9.11: This time, the greater resources allow Chuck to infer a precise type.

Chapter 10

Empirical validation of DDP

The **DDP** algorithm has been evaluated empirically. There are two claims that the experiments attempt to validate:

- The algorithm scales to produce useful results on real programs with hundreds of thousands of lines of code.
- Subgoal pruning gives significant improvements in the performance of the algorithm.

Additionally, the experiments attempt to determine good choices of the pruning threshold.

The first claim is the most interesting and is the bulk of the thesis. It shows that there is an effective algorithm for finding type information in large dynamic programs.

The second claim is that subgoal pruning is worth the complexity it adds to the algorithm. A possible alternative is to only allow pruning the primary goal; the algorithm would be simpler, but it is expected that the precision would decrease.

This chapter describes the experiments that have been performed, gives the results from those experiments, and analyzes those results.

10.1 Issues

10.1.1 Better versus good

Most researchers experimentally validate a program analysis by implementing it and then comparing one system that uses the analysis to another that does not. Such researchers might compare the results of the analysis directly to the result of another analysis. Alternatively, such researchers may modify an application to take advantage of information from the analysis and then compare the performance of the application when it does or does not use the analysis. For example, they might implement a dead code remover using information from the analysis, and then measure what percentage of a sample program is removed by the dead code remover.

Neither of these approaches work well for testing **DDP**.

First, there truly are no competing algorithms to compare against, as described in the related works section. It would be possible to implement competitors myself, but there would still be a question of whether my implementations were at fault instead of the general algorithm. No serious contender has been implemented for Smalltalk itself, and thus any existing algorithms would need some amount of adaptation. Challengers could continuously request variations and improvements on the adaptations, and the question would always remain whether the next improvement might in fact make the algorithm practical.

Further, I know of no existing applications, such as compilers, that can take advantage of type inference. It would be possible to implement such applications, but that requires a large amount of work.

To put it briefly, this analysis area is simply too new for comparative validation to be effective.

Thus, instead of comparing **DDP** to other algorithms, the experiments evaluate whether the algorithm performs *usefully well* for some applications, not whether it performs *better* than some other algorithm.

The next two sections discuss what usefully good performance would mean for **DDP**.

10.1.2 Performance of demand-driven algorithms

DDP, like all demand-driven algorithms (see Chapter 2), finds one fact or a small number of facts for each execution. In contrast, exhaustive algorithms find a complete set of facts about an entire program. Demand-driven algorithms are typically slower for analyzing entire programs, but faster for analyzing small portions of programs.

When measuring a demand-driven algorithm, it is the performance per fact inferred that matters. The experiment thus measures performance per inference instead of the performance for an entire program.

10.1.3 Performance of type-inference algorithms

There are two aspects of performance of a type-inference algorithm: *speed* and *precision*. An algorithm with better speed finishes more quickly. An algorithm with better precision produces inferences that are more specific, e.g., "x is a SmallInteger" has better precision than "x is a SmallInteger, a LargeInteger, or a Float".

Measurement of speed is straightforward: simply record the amount of time required for the program to complete. An algorithm that takes half the time as another performs twice as well as the other.

Measurement of precision might be difficult, at least in principle. What is a precise type inference? What does it mean for a type to be, say, *twice* as precise as another?

In practice, experience with **DDP** suggests that most results strike observers as either very precise or very imprecise, with uncertain cases being marked as imprecise. This strategy is consistent with the goal of verifying the algorithm to produce usefully precise results for program-understanding applications. This strategy does not produce a verified, objective measure of precision, but does give a *conservative* measure of precision that can be used to verify that the algorithm passes a certain threshold of precision.

The following specific rules were used to classify each type inferred for a variable:

- $\top$ is imprecise.
- {|UndefinedObject|} is precise. It means that the variable is never assigned a value, and thus it only holds `nil` when the program runs.
- Any type that is the union of {|UndefinedObject|} with a 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. Notice that this is the only rule where human analysis is required; the other rules leave no room for interpretation.
- If none of the above rules apply, then the type is imprecise.

10.1.4 Usefulness

Some correct type inferencers are trivial and useless. For example, an inferencer could report type $\top$ for every query it is posed. Since every value is within type $\top$, such an answer is always correct, and thus the inferencer is correct as well. Nevertheless, such an inferencer is useless. A compiler writer would never use such an inferencer even if it was fully implemented and only required a single function call to invoke. A tool author would never waste screen space on an "infer type" button that invoked such an inferencer. Correctness is not enough for a useful algorithm.

DDP is more sophisticated than this trivial algorithm, but is it truly useful? Perhaps it is equally useless, only in a more complicated way? How can one distance **DDP** from the trivial algorithms? Since it is impossible, as discussed in the previous section, to show that **DDP** is *better* than some other algorithm, one cannot simply show that **DDP** is better than some existing non-trivial algorithm. Instead, one must show that **DDP** is sufficiently *good* that it is non-trivial. The present work shows that **DDP** is sufficiently good that it may be called *useful*.

Usefulness is a sufficiently strong claim to establish **DDP** as one type inference algorithm that finds a non-trivial amount of correct type information. Given the long history of type inference efforts, this level of performance is high enough to establish **DDP** as a first algorithm in its domain to compare against.

Usefulness, however, is not a strong claim. It is not a claim that the tool will be useful for all purposes or even most purposes. In fact, **DDP** does not appear useful for dead code removal. It is also not a claim that **DDP** is extremely useful, but instead only that it is somewhat useful, even for applications where it is useful at all.

From the opposite point of view, to disagree with a weak claim is to make a strong claim in the opposite direction. The opposite of the claim sought in the present empirical work, is the claim that **DDP** is completely useless for all purposes. Readers should be careful before rejecting the weak claim of usefulness for some purpose, lest you commit yourself to accepting a strong claim of complete uselessness.

Trying to establish usefulness causes complications. One complication is that usefulness depends on the effort a particular user is making. A screwdriver is very useful for someone who wants to screw things together, but quite useless for someone who wants to prove a mathematical theorem. To address this complication, the present work examines multiple typical applications of type inference and evaluates the algorithm's usefulness for each of these applications. The hope, needed to satisfy the claim in the thesis statement, is to find **DDP** useful for at least one typical application.

Another complication is that usefulness is not a sharp criterion. Much like with beauty, wealth, and precision, there is no obvious threshold for usefulness. Different users will simply have different standards. To address this difficulty, this document invites readers to speculate on levels of performance they believe would be sufficient for a type inferencer to be deamed useful. Try make that decision before reading the final performance data; otherwise, you will lose some of your objectivity.

Different readers will choose different standards of performance. Some, doubtless, will choose a high enough standard that **DDP** does not meet it. Such readers must conclude that the present work merely moves the field closer to a useful algorithm, without yet achieving the grail of a usefully precise type inferencer. An effort is made, below, to specify thresholds that most readers will agree are sufficient.

Overall, usefulness is not a convenient criterion. Nevertheless, it is an important attribute of any tool, especially a tool that is claimed to be a first success in its (narrow) field. It is worth making the effort to address it. To contrast, it would be no improvement of the present work to omit discussion of the topic of usefulness, simply because it is difficult to talk about or because the conclusions are not as rigorously established as a proven mathematical theorem. It would be no improvement to focus all efforts on the clearest problems. Sometimes, important issues are hard to discuss.

10.1.5 Performance criteria for usefulness

This section provides some target thresholds we believe are sufficient to call a type inferencer *useful*. The reader is invited to choose thresholds of your own before reading the author's choices.

A threshold is specified for each of the following applications:

- Programmer queries. A human programmer trying to understand a program, asks the tool questions as they occur.

- Optimization of individual modules.

- Optimizations that require a small subset of the possible facts. For example, a compiler might want points-to analysis only for arrays that are indexed from within loops.
- Dead-code removal.

For the first three applications, the performance of the algorithm on an entire program is irrelevant. For the last, the performance on the entire program does matter, but for consistency the target will be re-calculated as a per-query target.

- Programmer queries: Each fact must require no more than one or two minutes to find. Preferably they require only a few seconds. Precise information must be found for at least one quarter of the queries, or programmers are unlikely to use the tool.
- Module optimization: A set of facts for an entire module requires no more than an hour, and preferably less than a minute. Precise information must probably be found for at least one tenth of the queries or so, or the analysis will not be worth the effort.
- Targeted optimizations: The requirements are variable. They depend on how small a subset of the facts the optimizer needs and on how large of a program fragment the optimizer is targeted at. Again, probably at least one tenth of the queries should yield precise information.
- Dead-code removal: A type inference for every message-send expression in the entire program must be found in no more than a week, and preferably no more than a day. Assuming there are on the order of one million send statements in the program, each query must be answered in one second. It is unclear how many queries need precise information; perhaps one tenth would be sufficient, though one would prefer a much higher precision.

These performance goals are not sharply defined—e.g., if it takes two hours instead of one to analyze a module, the algorithm is still worth something—but they give a rough idea of what level of performance is needed for a demand-driven algorithm to be useful for various applications.

10.2 Alternative experimental designs

There are a number of experiments that could be performed on **DDP**. Since these experiments are typical in the field of program analysis, it is worth discussing why those experiments have not been performed on **DDP**.

The next section describes the experiments that have actually been performed on **DDP**.

10.2.1 Comparison to competitors

A very common approach for experimentally testing a program analysis is to directly compare the performance of the algorithm to the performance of other algorithms that solve the same problem. If **DDP** performs better than the competitors, then it would show that **DDP** is performing well enough to be interesting.

As described in Chapter 2, there are no reported analyses for a dynamic language in such large programs, and there are no context-sensitive algorithms that even appear to scale. There are, however, context-insensitive algorithms that have linear complexity and thus should scale in principle. It would be possible to implement one of the linear-complexity algorithms and thus do a direct comparison.

The primary difficulty with this approach is that the linear-complexity algorithms are still linear in the program size in both time and memory. My efforts so far to modify such algorithms for Smalltalk and run them against a sample large program, have resulted in the machine paging constantly to disk before the algorithm even finished generating all of the constraints, on a machine with 512MB of RAM. More engineering work and better machines, might produce a practical implementation, but the required effort appears to be substantial.

An additional difficulty is that these algorithms are not described for Smalltalk. Thus, while the general approach of the algorithms transfer, some cleverness is still needed. As one example, there is no syntax in Smalltalk for instance creation; instead, one sends the `#new` message to a class object. This is no challenge at all for a context-sensitive algorithm, but without care, a context-insensitive algorithm would conclude that all senders of `#new` return the same type. Likewise, blocks are invoked by a message send, not by syntax, and those executions as well should get some care in a serious implementation. Thus, it requires considerable work and cleverness to transfer any of the existing algorithms to Smalltalk, and for all of that effort, it is unclear which algorithms will in fact produce results at all to compare against.

10.2.2 Comparison to competitors in other languages

Instead of porting program analyses to Smalltalk, an experimenter could port **DDP** to other languages. In particular, one could target the Cecil language [17], and thus perform a direct comparison against the mature analyses that are part of the Vortex compiler for Cecil. If **DDP** performs better than the other analyses, then the experiment would show that **DDP** is performing well enough to be interesting.

The first difficulty with this approach is that it again requires a substantial implementation effort. The experimenter must learn the alternative language thoroughly enough to perform analyses in it, adapt the analysis to work in that language, and then fully implement the analysis.

The second is that it renders the defense of the thesis less cohesive. It would be perfectly acceptable to this researcher to prove that **DDP** works in some other dynamic language, e.g. Scheme or Cecil. However, the rest of the defense speaks to the thesis that **DDP** works in Smalltalk. Most significantly, the proof of correctness (Chapter 8) refers to Smalltalk, and the programming tool (Chapter 9) is implemented in Smalltalk. Changing the thesis, requires not only re-implementing **DDP** itself, but also performing again most other components of the defense of thesis.

10.2.3 Performance for smaller programs

It would be possible to use smaller Squeak programs, to implement competitors known to be effective in smaller programs, and then to compare the performance of **DDP** to the competitors. If **DDP** produces precise results in smaller programs, then the experiment would show that **DDP** is effective in small programs. Additionally, the experiment would *suggest* that **DDP** would also produce precise types for larger programs. One would need to perform an additional experiment to show that **DDP** does complete in reasonable time for larger programs.

There are three difficulties with this approach.

First, it requires a substantial implementation effort. The competing algorithms are not implemented in Smalltalk, and thus they must be modified to work in Smalltalk and also implemented from scratch.

Second, this experiment does not stand alone. In order to learn from this experiment, one must perform an additional experiment to learn (hopefully) that **DDP** does terminate in reasonable time on larger programs. Without that experiment, then there is no evidence regarding **DDP**'s effectiveness on larger programs.

Finally, the experiment provides only indirect evidence about the desired thesis. This author is interested in larger programs than have been proven to be supported by any published algorithm, but the experiment reports results on small programs. If one has limited time for experiments, then surely one should seek an experiment that gives direct evidence.

10.2.4 Performance of applications

Instead of comparing the analysis against other analyses, it would be possible to use the analysis to improve some application such as a compiler or a dead code remover. Then, one could compare the performance of the algorithm with **DDP** to the performance it attains without it. If the applications perform significantly better when using **DDP** than when not using it, and if the applications do not take an unreasonable amount of time when they consult **DDP**, then the experiment would show that **DDP** is performing usefully well.

The main difficulty with this approach is that a substantial implementation effort is required. There is no optimizing compiler for Squeak, the dialect of Smalltalk used in this research. Writing an entire optimizing compiler is clearly an overly extreme effort if the only goal is to validate a program analysis. Porting to other Smalltalks is a significant effort, as well. Further, it is unclear which Smalltalk to port to. Cincom VisualWork[1] has a good runtime, but it is unclear whether its owner would let a researcher from the general public access that runtime system. Self [70, 62] has a good runtime, but the language is different enough to run into the porting difficulties described in the previous section.

Squeak does have a simplistic dead code remover. However, **DDP** as it stands is not organized for effective dead code removal, because the algorithm has been studied for the application of program understanding. Dead code removal requires analyzing all of the program that is live, and **DDP** per se is more efficient at targeting individual expressions. As discussed in Chapter 12, it remains future work to modify **DDP** to be effective in contexts where a large number of queries are being submitted to it. That effort is too substantial to perform merely for the sake of an experimental effort. Further, it does nothing to test **DDP** for use in program understanding tools, which is this researcher's primary interest in type inference.

Finally, Squeak does include several code browsers, and those code browsers include various queries used for program understanding. Some of these queries can be improved by using **DDP**, and one could perform an experiment to see how much, if any, those queries are improved if they use **DDP**. This experiment would produce the same strength of justification as the experiment actually performed, with the same amount of effort, and thus only smaller matters decide between them. I chose the experiment described below, because it gives more direct evidence, and because it seems like a better contribution to produce a tool that solves a problem not known to be solvable, than to improve on an existing tool.

10.2.5 Summary

In summary, all of these approaches require a substantial amount of extra implementation work. Additionally, most of the approaches cause difficulties with the thesis. Either they require a change to the thesis that would render the defense less cohesive, or they require a change to a thesis that is less interesting to this researcher.

10.3 Actual experimental design

This section describes the experiment actually performed. It directly addresses the two claims described at the beginning of this chapter.

10.3.1 The program code tested

The experimental executions include queries on Squeak 3.7, a Smalltalk system that, when the type inferencer is loaded, has 358,872 non-blank lines of code, 2485 classes, and 48,715 methods. The program includes a large variety of software such as a web browser, an Internet Relay Chat [50] client, a port of the Alice system [18] for end-user programming in three-dimensional spaces, and the platform-independent portion of the Squeak interpreter itself [38, 33].

The experiment infers a type for each instance variable in nine components of the program, as summarized in Table 10.1. A total of 765 variables are analyzed. The components cover a variety of application domains and a variety of authors.

The algorithm is given no information about where execution might begin or about which portions of the code base constitute an application or a module. Thus, the algorithm sees a single large 300,000-line program even though each query will analyze only a subset of the program.

[1] `http://smalltalk.cincom.com`

Name	Instance variables	Description
rbparse	56	Refactoring browser's parser
mail	73	Mail reader distributed with Squeak
synth	121	Package for synthesis and manipulation of digital audio
irc	114	Client for IRC networks
browser	32	Smalltalk code browser
interp	173	In-Squeak simulation of the Squeak virtual machine
games	135	Collection of small games for Morphic GUI
sunit	10	User interface to an old version of SUnit
pda	51	Personal digital assistant

Table 10.1: The components of the program analyzed.

10.3.2 The trials

Each trial uses the implementation to infer a type of one variable. The trials vary the following parameters exhaustively:

- They choose a variable from the instance variables in the packages that are tested.
- They choose a pruning threshold that is either infinite or that ranges among 12 values from 50-10,000. If an infinite threshold is chosen, then no pruning is performed; instead, the algorithm is executed for 5 minutes on each query. If no result has been found within the time limit, then a result of $\top$ is returned. This variation is a "drop dead" timeout.

For each trial, the amount of time and the inferred type are recorded. Followup human analysis classifies each inferred type as precise or imprecise using the approach described above.

10.3.3 The machine

The trials are executed on a machine with an Intel Celeron CPU, clock speed 2.40 GHz, and 512 MB of RAM.

10.4 Summary of results

The measured speed of the inferencer is tabulated in Table 10.2 and summarized as a graph in Figure 10.1. The measured precision of the inferencer is tabulated in Table 10.3 and summarized in Figure 10.2. An overall speed-precision performance envelope is shown in Figure 10.3.

The following types, which are obviously precise, comprise 93% of the inferences that were classified as precise:

- (57.6%) ⦃C⦄ $\sqcup$ ⦃UndefinedObject⦄, for some class C.
- (30.5%) ⦃UndefinedObject⦄, i.e., the variable is never initialized from the code.
- (9.4%) ⦃True⦄ $\sqcup$ ⦃False⦄ $\sqcup$ ⦃UndefinedObject⦄
- (5.4%) ⦃SmallInt⦄ $\sqcup$ ⦃LargePosInt⦄ $\sqcup$ ⦃LargeNegInt⦄ $\sqcup$ ⦃UndefObject⦄[2]

[2]Class names have been abbreviated.

	50	150	300	500	1k	1.5k	2k	3k	4k	5k	7.5k	10k	inf
rbparse	0.9	2.0	4.0	5.6	10.1	15.0	20.5	27.3	48.0	43.9	73.2	120	155
mail	1.8	3.6	7.9	8.2	14.4	19.6	27.3	37.1	51.0	85.9	89.6	140	219
synth	1.0	3.5	5.8	7.9	12.5	21.3	25.1	35.5	41.9	55.8	123	153	375
irc	0.45	1.5	2.0	2.9	4.8	7.0	6.9	10.6	42.7	41.4	45.7	48.1	62.3
browser	0.82	3.6	6.7	10.1	15.0	20.4	27.8	68.3	52.4	60.1	133	167	207
interp	0.46	1.2	5.1	4.8	8.4	11.3	16.3	24.0	27.9	31.8	51.8	63.5	232
games	0.82	2.2	3.5	5.7	10.8	15.9	19.4	27.8	34.0	36.6	62.1	73.9	161
sunit	1.0	1.5	4.4	4.9	11.2	14.4	11.4	29.6	31.9	36.6	52.1	76.1	47.4
pda	0.68	2.9	5.1	7.7	18.2	27.3	39.5	50.7	67.0	72.0	115	199	271
Overall	0.83	2.3	4.7	6.0	10.6	15.6	20.2	29.7	40.8	47.8	77.0	102	209

Table 10.2: Speed of the inferencer. Entries give the average speed in seconds for inferences of instance variables in one component, using the given pruning threshold. For example, when the pruning threshold is 50, the `rbparse` package requires an average of 0.9 seconds to infer a type for one of its variables. The "inf" column is for executions where no pruning was performed, and instead the implementation was given 5 minutes per variable to infer a type if it could. 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	150	300	500	1k	1.5k	2k	3k	4k	5k	7.5k	10k	inf
rbparse	25.0	28.6	28.6	28.6	30.4	30.4	32.1	32.1	33.9	32.1	33.9	33.9	30.4
mail	28.8	34.2	37.0	38.4	41.1	37.0	37.0	39.7	37.0	41.1	37.0	38.4	31.5
synth	28.1	31.4	38.8	38.8	38.8	40.5	40.5	43.0	43.8	43.0	46.3	47.1	34.7
irc	69.3	72.8	75.4	76.3	77.2	78.1	79.8	81.6	81.6	80.7	79.8	80.7	77.2
browser	9.4	12.5	12.5	12.5	15.6	15.6	15.6	15.6	18.8	15.6	12.5	15.6	9.4
interp	17.9	21.4	22.0	22.0	22.0	25.4	29.5	31.8	31.8	31.2	31.8	31.2	29.5
games	51.1	51.1	56.3	60.0	60.0	62.2	71.1	74.1	73.3	73.3	74.1	74.8	61.5
sunit	40.0	50.0	60.0	50.0	60.0	60.0	60.0	60.0	60.0	60.0	60.0	60.0	60.0
pda	19.6	21.6	23.5	23.5	25.5	33.3	33.3	35.3	35.3	37.3	35.3	37.3	21.6
Overall	34.6	37.6	40.7	41.5	42.4	44.1	47.0	49.1	49.1	49.0	49.1	49.8	42.3

Table 10.3: 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. For example, when the pruning threshold is 50, the `rbparse` package gets precise types inferred for 25.0% of its variables. The "inf" column is for executions where no pruning was performed, and instead the implementation was given 5 minutes per variable to infer a type if it could. As in Table 10.2, the "overall" entries are averaged across inferences, not averaged across the averages in the table.

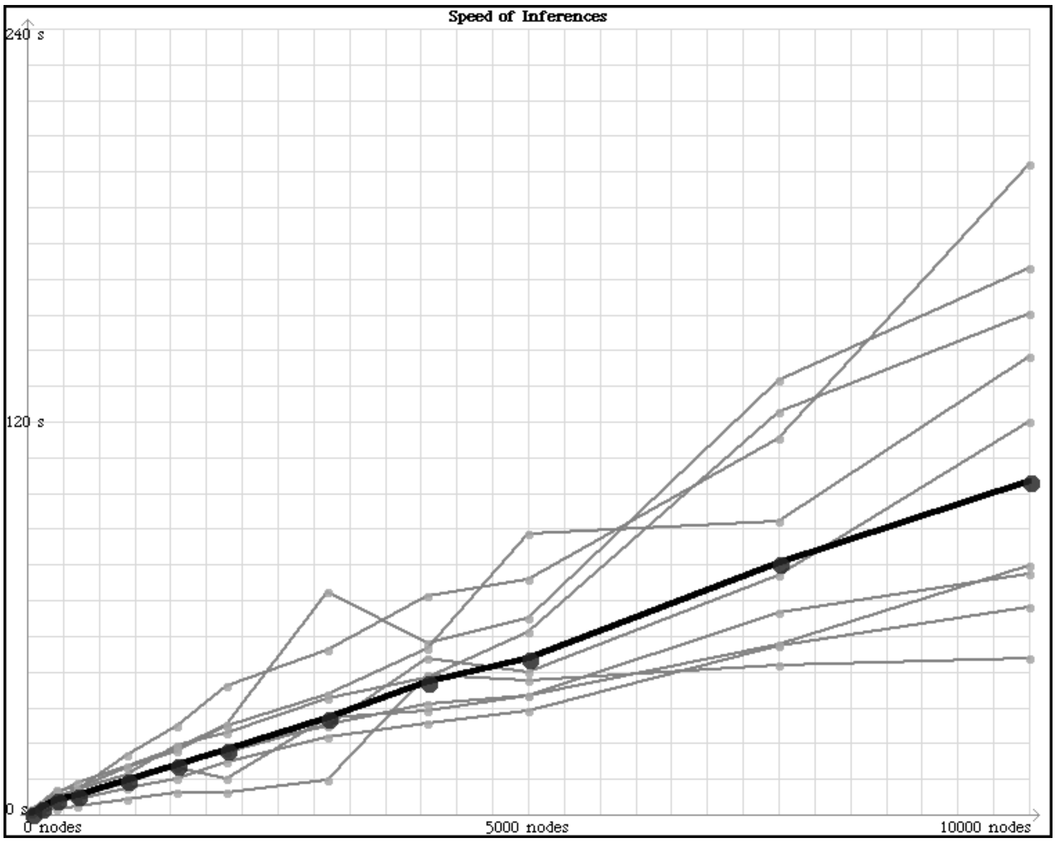


Figure 10.1: Graph of the inferencer's speed. The horizontal axis is the pruning threshold, and the vertical axis is the average time required for each inference. The thick black line gives the overall average, while the gray lines each give an average for one component.

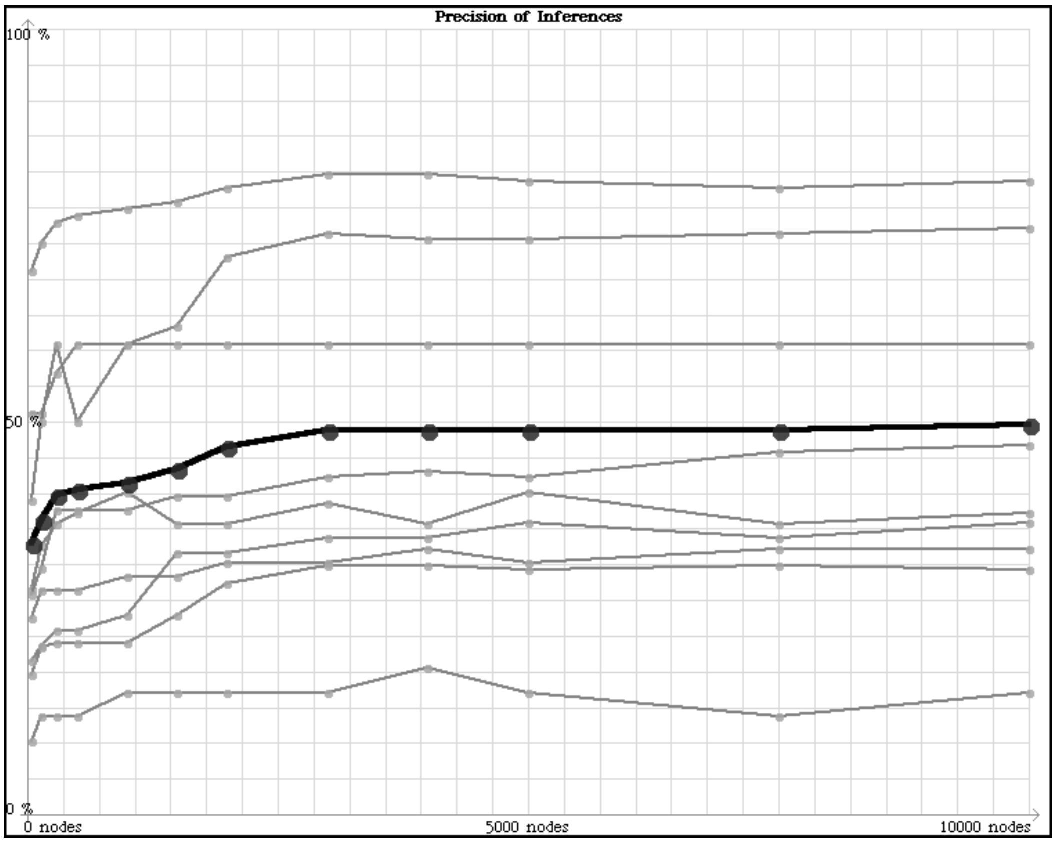


Figure 10.2: Graph of the inferencer's precision. The horizontal axis is the pruning threshold, and the vertical axis is the percentage of the inferences hand-classified as precise. The thick black line gives the overall percentage, while the gray lines each give a percentage for one component.

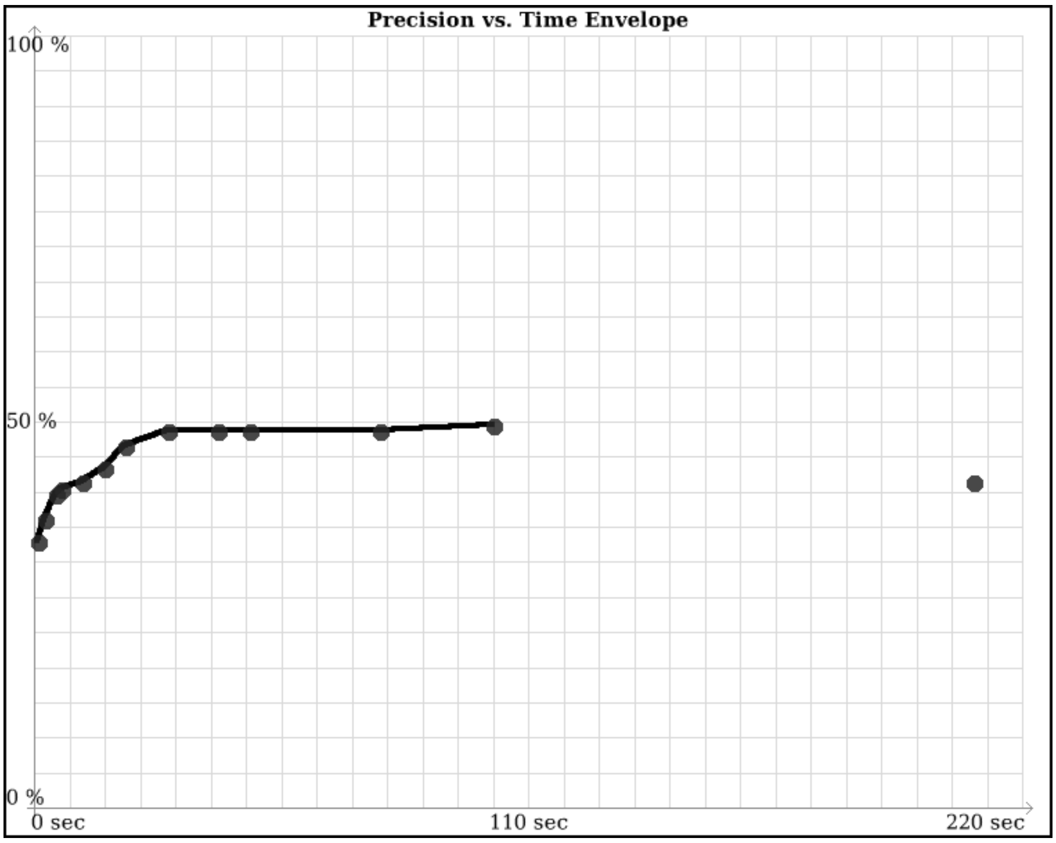


Figure 10.3: Graph of speed-precision envelope. This graph combines the thick, black summary lines from Figure 10.1 and Figure 10.2, thus showing the overall performance envelope of the **DDP** prototype. Additionally, the overall performance of the drop-dead variation, i.e. the “inf” columns from Table 10.2 and Table 10.3, is summarized by the single data point to the right of the the main data in this graph. Note that the drop-dead variation is on the graph and is a practical algorithm for some applications, but its performance is worse than the performance envelope of **DDP**.

10.5 Analysis and conclusions

The experimental results lead to a number of conclusions.

First, the level of pruning matters. Varying the pruning threshold causes the precision to vary from 34.6% to 49.8%, and the average time required per inference to vary from 0.83 seconds to 209 seconds. The pruning threshold is certainly an effective knob for tuning the algorithm, both for speed and precision.

Second, there are two pruning thresholds that seem to give a useful level of both speed and precision. A tight pruning threshold of 50 gives high speed (0.83 seconds) and high enough precision to be useful (34.6%). Such a choice would be good for all applications described in subsection 10.1.5: program understanding, targeted optimization, and (barely) dead code removal. Such a choice seems especially sensible for program-understanding tools, where a user is waiting and every second matters. A threshold of 2,000 gives reasonable speed (20.2 seconds) and a higher precision (47%). Such a choice would be good for an optimizing compiler, which can often afford to spend several seconds if it gives a better compilation. Higher thresholds than 2,000 continue to slow the inferencer down but do not give much higher precision; the maximum precision attained in the experiments was 49.8% percent. No setting of the threshold gives performance sufficient for practical dead-code removal.

Third, subgoal pruning is valuable in general. Goals after the first few thousand do not substantially increase precision (Figure 10.2), but they steadily increase time (Figure 10.1). Non-pruning demand-driven algorithms implicitly allocate a number of goals only limited by bounds such as the size of the program and the richness of the domain of goals.

Fourth, subgoal pruning appears to focus the algorithm's effort more effectively than a time-based timeout. The "inf" columns of Table 10.2 and Table 10.3 show that a timeout approach can find a large number of precise types (42.3%), but it requires a large amount of time to find them (209 seconds per query). One could instead use subgoal pruning with a threshold of 1000, and thus find slightly more types that are precise (42.4%) while requiring an order of magnitude less time per query (10.6 seconds). Alternatively, one could choose a pruning threshold of 10,000, thus finding significantly more types that are precise (49.8%) while requiring roughly half of the time per query (102 seconds). The difference is shown visually in Figure 10.3. Thus, drop-dead timeouts provide an alternate but inferior technique to subgoal pruning for making context-sensitive data-flow analysis practical in a Smalltalk-like setting.

Finally, there is one anomaly in the data. Some of the trials with no pruning require more time than the 300-second maximum. The number of such trials is not yet known, and the amount of overrun is not yet known, either. It appears that the implementation, for some reason, does not always stop immediately at 300 seconds. A better implementation would fare better in the no-pruning trials. The data should be re-analyzed, with all timings greater than 300 seconds being replaced by the 300 seconds that an improved implementation would have achieved.

10.6 Informal notes

Perusal of the full experimental results, published separately, provides various intuitions about the performance of the system. We briefly share some of those intuitions in this section. Much of the future work in Chapter 12 consists of exploring these intuitions more fully.

A large amount of code is not overtly polymorphic. The best-performing packages—IRC, Morphic Games, and SUnit—include a large number of variables that are ultimately assigned an expression such as "`FreeCellBoard new`". While program-analysis researchers enjoy considering sophisticated code patterns, a large amount of practical code uses simpler idioms. The present work thus reemphasizes an observation from analysis researchers tracing back to at least Knuth's study of typical FORTRAN programs [44].

The best-performing package, IRC, additionally has many unused variables. Whenever **DDP** is queried for the type of an unused variable, it instantly infers a type of {UndefinedObject}. The presence of unused variables further emphasizes the above observation about simple code being surprisingly common.

On the other end of the spectrum, the Browser package provides a number of examples that use both integer arithmetic and round-trips through the highly polymorphic GUI libraries, in particular the class PluggableListMorph. In principle **DDP** has enough polyvariance to be effective in this case, but for some reason the analysis is not succeeding. Therefore, the browser package provides excellent example code to investigate for future improvements of the justification rules.

Finally, it should be noted that examples appearing to be simple to a human often require at least one surprisingly sophisticated and precise step in the derivation. A salient example is the `connection` variable of class `IRCChannelListBrowser`. The values stored into this variable are ultimately created by either the expression `self` in a method of class `IRCChannel` or the expression "`IRCConnection new`." Both of these expressions are trivial to analyze, but the path between those expressions and the `connection` variable include the parameter of a method named `initialize:`.

Finding the type of the parameter to this `initialize:` method requires analyzing 45 potential senders of `initialize:` and determining that only one of them is feasible. If the analyzer failed in this step of the derivation and considered 2 of the 45 to be feasible, then the inferred type would almost certainly at least double. Additionally, each additional feasible sender adds an extra chance for the analyzer to fail and return type $\top$—a substantial risk in an analysis that finds precise types for top-level queries roughly 30% to 50% of the time.

10.7 A pruning schedule for interactive use

The experimental results shed light on a new pruning schedule for interactive use such as in Chuck (Chapter 9). The trials all use a simple, fixed pruning threshold. For interactive use, it appears useful to instead have a *pruning schedule*, where the pruning threshold decreases as time goes on.

With the algorithm from section 4.8, the choice of fixed pruning threshold gives a rough control on the time required and the precision obtained. This control is loose. For example, as reported above, a threshold of 3000 nodes yielded an average time of 30 seconds per query, but the slowest of those queries required over 10 minutes.

For interactive use, these occasional large response times are not acceptable. We would prefer to provide consistently fast responses even if the responses are not as precise as possible. A crude way to obtain consistently fast responses would be simply to halt the algorithm if a response has not been found within some time limit and report failure. That is, run the analysis, and if it requires more than, say, five seconds, terminate it and report that no information was found.

A more graceful degradation of precision than this approach may be obtained by taking advantage of the structure of the **DDP** pruning algorithm. The tool begins by using a pruning threshold of 3000. If no result has been found within three seconds, then the pruning threshold is decreased to 50 and the algorithm is given two more seconds to complete. Usually the algorithm finishes in a fraction of a second with a pruning threshold of 50, but in the rare case that it requires two or more seconds, the algorithm is terminated after all—by lowering the threshold to 1—and the system reports that no information was found.

Further analysis of data from the above experiments show that the cruder approach, that of using a threshold of 3000 and then stopping after five seconds, yields an answer—precise or imprecise—to 40% of type queries. The remaining 60% would necessarily have to be answered with the maximal type, $\top$, because **DDP** does not provide sound information if it is halted early. The more gradual approach, with a threshold of 3000 for 3 seconds followed by 50 for 2 seconds, finds answers to a total of 94% of the queries: it answers 37% in the first three seconds, and 57% after reducing the threshold to 50. Both approaches have a maximum execution time of five seconds, but gradual reduction of the threshold yields a complete analysis of many more queries (94% versus 40%). Further, the expected analysis time for the gradual-reduction schedule is 2.6 seconds per query, versus 3.3 for the drop-dead schedule.

With this pruning schedule, the time required per query does not depend on the speed or load of the underlying machine. All queries finish in five seconds. Instead, the speed and load on the underlying computer affect the *quality* of results that **DDP** produces. A slower machine will still finish each query in five seconds but will produce less precise results.

- total queries: 765
- at 3000 threshold:
  - queries finishing in under 3000 ms: 281
  - total time for those: 110946 ms
  - queries finishing in under 5000 ms: 305
  - total time for those: 206968 ms
  - queries that take over 3000 ms at threshold of 3000: 484
  - queries that take over 5000 ms at 3000: 460
- at 50 threshold:
  - queries that are under 2000 ms at threshold 50 and also over 3000 ms at threshold 3000: 438
  - total time for these: 365521 ms
  - queries that are both over 3000 ms at threshold 3000 and over 2000 ms at threshold 50: 46
- drop-dead at 5 seconds should require 3277 ms/query:

$$\frac{206968 + 460 * 5000}{765}$$

- gradual decay should require 2641 ms/query:

$$\frac{110946 + 365521 + 484 * 3000 + 46 * 2000}{765}$$

Table 10.4: Calculation of expected time for the gradual-reduction pruning schedule.

This schedule has been designed according to experimental results, but it has not been experimentally tested. It remains future work to determine whether this schedule produces the high level of precision we expect based on the available data thus far.

Chapter 11

Proposed language changes

A fundamental goal of the present research project has been to perform analysis in an unmodified, extremely dynamic programming environment that makes no concessions to ease of analysis. This goal is valuable, because it demonstrates that, given suitably rich *dynamic* semantics including memory safety, type-tagged data values, and (dynamic) type safety, it is possible to perform a substantial amount of data-flow analysis. Nevertheless, there is no need for future software developers to work with a language at this extreme. This chapter briefly explores some language changes and research directions that seem likely to improve the performance of data-flow analysis without harming the overall character of the programming environment.

Initialized Variables Variable bindings in Smalltalk are automatically initialized to hold the special value `nil` whenever there is no other value to give them. This automatic initialization applies to all forms of variables other than parameters; parameter bindings are created at the time of a message send and thus their initial values are specified in the actual arguments of the message. Automatic initialization is necessary, given the rest of the language definition, in order to provide the valuable property of memory safety. It is impossible in Smalltalk to access memory in a way that violates the expected memory model. If variables were not automatically initialized, then it would be possible for a variable to hold an arbitrary bit pattern and for a program that accidentally uses that pattern as an object reference to corrupt memory.

Automatic initialization has a negative effect on type inference, however, as pointed out previously by Agesen [3]. A correct type inferencer must declare that all variables other than parameters can hold `nil`, i.e. type ⟦UndefinedObject⟧. This is an immediate loss of precision if the variable is defined to a more useful value and the automatically supplied `nil` is never used. Additionally, it can cascade into other precision losses for data-flow queries that depend on type queries on variables. Looking ahead, a keen example of the problem is the new type-specific flow goals that are forced to be added in Chapter 13.

A property that can be exploited to improve this situation, however, is that many variables in typical programs do not have the automatic initialization value read from them. Every time such a variable is bound, it is assigned a new value before the program ever reads from the new binding. Automatic initialization is important for safety, but it is frequently not needed for expressiveness.

Agesen exploits this property by performing an extra analysis before the type-inference proper in order to statically detect many cases where a variable's automatically initialized value is not used. He reports a substantial increase in precision as a result [3].

An alternative solution that is worth exploring is to amend the language to allow initializers with variable declarations. Instead of merely declaring that a block has a temporary variable named `foo`, a block can simultaneously declare the variable and give it an initial value of, say, 0. This approach does not increase the verbosity of programs but does allow a safe mechanism for a type inferencer to avoid polluting more variables with type ⟦UndefinedObject⟧. This approach is used in a wide variety of languages including C [43], Java [30], and Ada [65], and we believe it would be a net improvement to add this feature to Smalltalk.

Soft Types Soft types [16] provide a mechanism to add the benefits of static types to a language without removing the dynamic character. Users can *optionally* annotate variables with types, and static tools can use whatever types the user has provided to gain various benefits such as error checking and improved compilations.

While soft typing is frequently promoted as a mechanism for error checking and improved compilation, it also has advantages for type inference. The type annotations provide upper bounds on the types that can be inferred. Any type query on a type-annotated variable can use the annotated type as an upper bound on the inferred type. As a particularly interesting case, a *pruned* type query does not need to infer a type of $\top$, but instead can infer a type equal to the annotated type.

Modules Module systems provide a number of techniques that can potentially reduce the number of feasible data-flow paths. The intuition is that most data- and control- flow occurs *within* modules, i.e. that there is relatively little flow *between* modules.

To achieve this advantage, the module system must provide some form of restriction on communication between modules. Static types at the module interfaces provide one mechanism for at least narrowing the communication channels that cross module boundaries. Adding static types to module interfaces does cause a dynamic language to become more static, but the dynamic character of the language could still be maintained for work that is within modules.

Another promising mechanism is that of ownership types [13, 7]. Ownership types provide statically checkable guarantees about the scope to which objects can flow. For example, ownership types allow checking the property that an OrderedCollection's internal array object only flows to the methods and instance variables of class OrderedCollection—that is, the array is *owned* by the OrderedCollection that uses it. It would appear that ownership types could be just as beneficial for limiting feasible flow paths for module systems as they are for individual classes.

It is a challenging research project to develop a module system that both maintains the dynamic character of Smalltalk while obtaining sufficient static restrictions that analyzers can benefit. Nevertheless, this suggestion is included in this chapter, notwithstanding that the rest of the chapter gives suggestions for straightforward improvements. Given the state of the art today, an suitable module system appears both feasible and useful.

Deployment-Time Interpreter A limited-strength deployment-time interpreter for Smalltalk would improve the reliability of results inferred by any static analysis. Static analyzers for Smalltalk must necessarily assume that extremely reflective features of the language will not be used at deployment time. If, for example, a program reads a string from the user and then recompiles methods depending on the contents of that string, then an analyzer has little hope of making a safe prediction about program behavior.

These reflective features are most frequently used by the development tools, not by applications themselves. Thus, it should prove useful for many applications to have a limited-strength version of the interpreter that does not allow those features to be used. Such an environment could even be built within a standard full-strength development environment. Developers could then test and deploy their code in the limited version of the language while performing the rest of the program development with the reflective power of the full language.

Chapter 12

Future work

This research reopens a static analysis problem that was widely suspected to be intractable. By proving the problem tractable and by providing an approach for solving it, the work opens a variety of future work.

12.1 Other languages and dialects

The current implementation is in the Squeak dialect of Smalltalk. The general approach of **DDP** should work, however, in a variety of Smalltalk dialects and in a variety of higher-order languages. It would be valuable to implement **DDP** in other Smalltalk dialects, and to adjust **DDP** for other languages entirely. Even statically typed languages can use the general approach and perhaps have better data flow information inferred. Intuitively, **DDP** should be effective in this wider variety of contexts, but one cannot be certain until it is tried.

12.2 Exhaustive analysis

It is sometimes useful to perform an *exhaustive analysis* of an entire program, but **DDP** is not designed to efficiently do so. **DDP** does allow for exhaustive analysis, simply by repeating the analysis on every variable and expression in an entire program, but this approach is probably less efficient than is possible. Much information would be calculated but then discarded; the subgoals of each target goal produce useful and true data-flow judgements but those judgements would be discarded.

For an efficient exhaustive analysis, it is desirable to keep old results and to reuse them in later queries. 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, before a judgement is reused, it is important to consider how close to the target the goal was.

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 without much loss in precision.

12.3 Pruning

The pruning approach implemented thus far is simple. While a simple technique gives a better validation of the abstract algorithm in general, the overall performance of a concrete algorithm is affected by the choice of pruning strategy. A good choice of pruning algorithm is a problem in artificial intelligence, and predictably there is a large area of investigation possible.

One specific idea that could be immediately explored is the following: some dependencies are stronger than others. For example, if one type judgement is required to have a supertype of another type judgement's type, then the two judgements are in a strong dependency—pruning one judgement would effectively cause the other to be pruned as well. If one type judgement merely influences the call graph, which in turn influences another type, then there is a weak dependency between the types. In the first case, pruning one judgement will effectively prune the other, so strong is the dependency. In the latter case, however, the dependency is so weak that the pruning may even have no effect at all. A better pruning algorithm would consider direct dependencies much more important than other dependencies. The occasional 4-times penalty described in section 4.8 is a simple example of this general refinement.

Another direction to investigate, related to pruning, is the character of the goal pool for typical problems. For example, such investigation could help find a good threshold size to use for a particular program. Currently, little is known about the goals and their dependencies, and thus guesses about the overall algorithmic strategy can only be evaluated by implementing them and trying them.

12.4 Other analysis problems

The present work studies *type inference* in the language *Smalltalk*. The general approach of **DDP**, however, appears promising for other problems and for other programming languages. It would be interesting to learn whether the general approach is effective more widely.

Type inference is an interesting problem for many languages. Probably **DDP** is effective in other dynamic languages such as Self and Scheme, and it would be interesting to verify that it is. Perhaps **DDP** is useful in static languages such as Java and ML, though only experimentation can say.

Further, there are other data flow problems that the approach might help with. Other data-flow analyses, such as alias analysis [67] and binding-time analysis [36], seem particularly promising.

Finally, the present work has used type inference based on CPA contours. It would be interesting to systematically analyze which other type inference approaches can be adapted to **DDP**. Probably, there are combinations that make sense. The current pruning approach of **DDP** choose between two extremes for each goal: either a precise CPA-based analysis, or an imprecise conservative analysis. Likely, other analysis approaches could be used as intermediate pruning levels.

12.5 Applications

Finally, it would be valuable to try other applications of type inference than program understanding. Compiler transformations and dead code removers would be good applications to try. They would both be useful and interesting tools in themselves, and they would both give alternative objective measures of the analysis's effectiveness. These extra object measures would be useful for guiding further development of the analysis itself.

Chapter 13

DDP/CT: Extending DDP with source-tagged classes

This chapter describes an extension to **DDP** called **DDP/CT**. The extension uses the concept of *source-tagged classes*, or *source tags* to support analysis in the face of *data polymorphism*. This extension has been implemented, but it has not been empirically validated and it has not been proven correct. Thus, this chapter describes part of the research frontier for the type inference work as described in this document.

Data polymorphism is, in Agesen's words, "the ability of a slot (or variable) to hold objects with multiple object types" [3]. 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 a vector of integers and a vector of strings.[1] As Agesen pointed out, data polymorphism can induce significant loss of precision 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 any 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 same 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.

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 in some fashion into the types (Vector, l_1), ..., (Vector, l_n) for some sequence of discriminators $l_1 \dots l_n$. The difference is shown in Figure 13.1 and Figure 13.2. 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 13.2, whereas a poor one leads to a flow graph like that in Figure 13.3.

For relatively static languages (such as Java [30]), an effective partitioning strategy is that of Wang and Smith's **DCPA** algorithm: subdivide objects according to which `new` expression instantiated them [72]. This approach yields a true partition because every object must have been instantiated by exactly one `new` expression; objects created by line 134 of the source code must be different from objects created by line 431.

For extremely dynamic languages such as Smalltalk, however, this approach is ineffective. 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 when a new object is created at run time, the classes are typically passed through a sequence of regular methods until arriving at the actual `basicNew` invocation. Since there is only

[1]In Hindley-Milner type inferencers, this facility is described by the term *parametric polymorphism* [15]. Since the analysis we are describing concerns an object-oriented language, and since the analysis is an intellectual descendent of Agesen's, we hew to the term data polymorphism.

[2]For clearer exposition, we are ignoring the existence of a small handful of such methods.

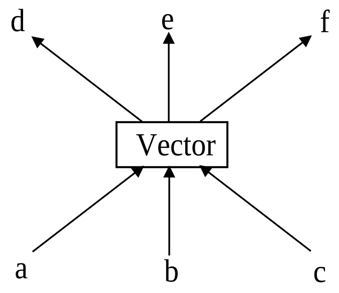


Figure 13.1: Without data-polyvariant analysis, a type inferencer intermixes all flows that go through a class's instance variables. For example, according to this analysis-level flow graph, objects in flow position `a` might flow to any of `d`, `e`, or `f`.

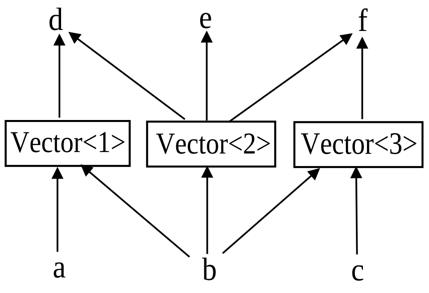


Figure 13.2: If instances of a class can be partitioned, then a type inferencer can, to an extent, segregate flows through the class's instance variables. For example, objects in flow position `a` can now be seen to flow to `d` but *not* to positions `e` or `f`. On the other hand, flow from `b` still reaches three positions. This partitioning improved the flow analysis of `a` and `c` but not of `b`.

one true `basicNew` method in the program, this kind of partitioning is trivial and unhelpful.

This chapter describes a partitioning strategy that achieves the effect of **DCPA**'s strategy in Smalltalk, even though Smalltalk programs only have a single instantiation point. Note that it is described in terms of the full Smalltalk language. Because Mini-Smalltalk includes a syntactic `new` statement instead of a `new` method, it does not capture enough of full Smalltalk to support describing the **DDP/CT** extensions.

13.1 Extensions

DDP/CT includes a number of individual extensions to the base **DDP** algorithm:

1. It extends the type system to allow class types to be subdivided using *source tags*.
2. It adds a new kind of goal, the *inverse type goal*.
3. It adds a second *solution strategy* for answering senders goals that uses inverse type goals.
4. It adds a new kind of goal for finding the *type of array elements*.
5. It *augments flow goals* so that they can trace the flow of just those objects within a specified type.

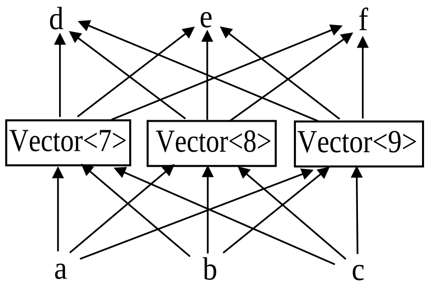


Figure 13.3: This poor choice of partitioning leads to a more expensive analysis with no improvement in precision. As in Figure 13.1, the analyzer will predict that flow from any of `a`, `b`, or `c` can reach any of `d`, `e`, or `f`.

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 instantiation 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.

13.1.1 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 **DDP/CT**, then, is to attach a *source tag* to all distinct references to a class in the source. Figure 13.4 and Figure 13.5 depict the general idea. This is a static or abstract analog to the dynamic "taint" attribute used in Perl for security purposes. Each location in the program text where a 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.

Source-tagged classes provide 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.

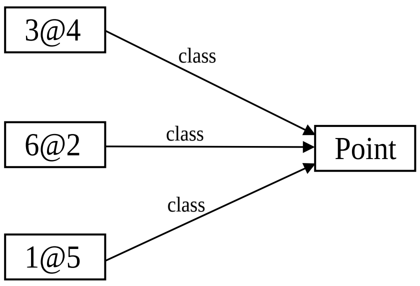


Figure 13.4: The semantics of Smalltalk are that classes are shared and singular. All instances of class Point (`3@4`, `6@4`, etc.) have a reference to the same class object.

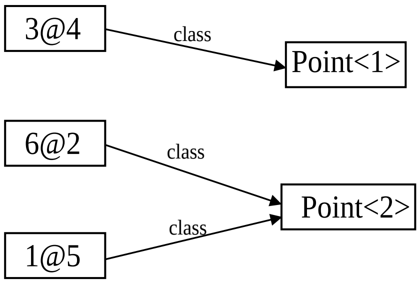


Figure 13.5: For **DDP/CT**, we can imagine that there are multiple copies of each class, one for each location the class is mentioned in the source code. Each copy is indistinguishable to regular program code—even the equivalence operator (==) does not distinguish them—and thus the compiler does not need to explicitly represent the tags. The tags are present in the abstract semantics, however, and can be reasoned about by a type inferencer.

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 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. Instead of solving such a goal with a simple class type as the base **DDP** would, **DDP/CT** solves the goal with a tagged class type.

13.1.2 Inverse type goals

A *inverse type goal* requests a flow position that includes all program locations that could produce an object of a specified type. 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 metaclass or not.

If C is a regular class and not a metaclass, then $\{\!|C, l|\!\}$ includes objects that were created by the `basicNew` method. 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 $\{\!|\mathbf{mclass}(C), l|\!\}$. We use "$\mathbf{mclass}(C)$" 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.

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

If C is a metaclass, then $\llbracket C, l \rrbracket$ 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 $\llbracket$**rclass**$(C), l\rrbracket$. 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 $\llbracket$**rclass**$(C), l\rrbracket$.

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. An inverse type query on such a class always returns position $\top_{fp}$.

13.1.3 Senders goals

Recall from Chapter 6 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 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 $\top$). 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.

13.1.4 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** provided no strategy for finding such types. **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 $\top$.

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.

13.1.5 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 depend.

The biggest problem appeared to be that **DDP** would trace flow paths that are feasible in principle but 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 subgraph of completely irrelevant program locations to be added to the potential flow from the original variable.

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** 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 solution strategy is then free to ignore methods to which `nil` flows but arrays do not.

13.2 Example Code Fragments

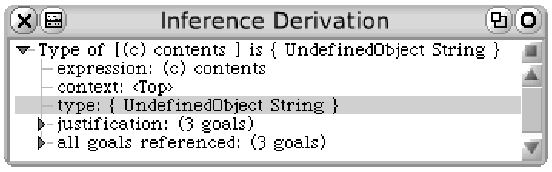
Figure 13.6 shows some example code that is data polymorphic. Class ValueHolder is a standard Smalltalk class used to hold an arbitrary value—it is a simple "cell" object. The internal value is set using the `contents:` method, and fetched using the `contents` method. The example code creates two value 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 Figure 13.7 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 13.6 in order to demonstrate an extent and a limitation of **DDP/CT**'s effectiveness. In Figure 13.8, the class ValueHolder is stored into variables `vhclass1` and `vhclass2` before being instantiated. This is an example of Smalltalk's reflective abil-

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

other contents.
c contents
```

Figure 13.6: 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 13.7.



```
Type of [(c) contents ] is { UndefinedObject String }
  expression: (c) contents
  context: <Top>
  type: { UndefinedObject String }
  justification: (3 goals)
  all goals referenced: (3 goals)
```

Figure 13.7: **DDP/CT** successfully infers that value holders assigned to `c` from Figure 13.6 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.

```
| 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 13.8: A variation of the code in Figure 13.6. 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 13.6.

ity 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 13.9 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.

Data-polymorphic analysis is especially useful when it is applied to resolving separate uses of collection types. A simple example is shown in Figure 13.10. 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 13.11. 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 13.12. 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 13.12 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 13.13.

13.3 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 13.14 shows the code from Figure 13.6 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 13.15 shows the goal graph that **DDP/CT** generates when it is asked to find a type for the expression "`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.

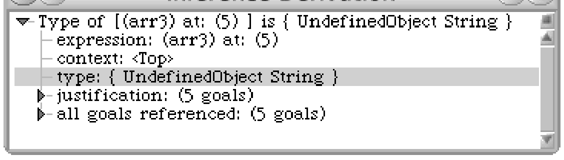
Goal G3 requests the type of the variable `c`. **DDP/CT** finds the type of a variable by finding the types of all

```
| 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 13.9: Another variation of the code in Figure 13.6. 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.

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

arr2 := arr.
arr3 := arr2.

arr3 at: 5
```

Figure 13.10: Retrieving elements from an array. Data-polyvariant 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.



```
▼ Type of [(arr3) at: (5) ] is { UndefinedObject String }
   expression: (arr3) at: (5)
   context: <Top>
   type: { UndefinedObject String }
 ▶ justification: (5 goals)
 ▶ all goals referenced: (5 goals)
```

Figure 13.11: The analyzer succeeds on the example in Figure 13.10.

expressions assigned to the variable. In this case, there is only one assignment, the expression `ValueHolder`$_1$ `new`, and so the type found for `c` is copied directly from the type found for `ValueHolder`$_1$ `new`.

The type for `ValueHolder`$_1$ `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 `ValueHolder`$_1$. Instead of being inferred as type ⦃ValueHolderclass⦄ as it would be in **DDP**, it is given a type of ⦃ValueHolderclass, 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 ⦃ValueHolderclass, 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 ⦃ValueHolderclass, 1⦄. To answer G12, **DDP/CT** creates goal G13 to find all expressions that hold an object of type ⦃ValueHolderclass, 1⦄.

Goal G13 is an inverse type query that finds all program locations holding a value of type ⦃ValueHolderclass, 1⦄, a type for a metaclass. **DDP/CT** immediately includes expression `ValueHolder`$_1$ because it evaluates to `ValueHolder class` with a source tag of 1. **DDP/CT** immediately *excludes* all other `ValueHolder` expressions, e.g. `ValueHolder`$_2$, 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.

13.4 Multi-level source tags

Factory design patterns [26] present an extra challenge to data-polymorphic analysis. A typical factory method is shown in Figure 13.16. This method provides a useful level of indirection—subclasses might override this method, and different platforms might replace the method outright. Unfortunately, the very indirection that motivates the design pattern circumvents the strategy 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 13.17. 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 13.18.

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.

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 13.17, resulting in the precise type inference shown in Figure 13.19.

13.5 Related work

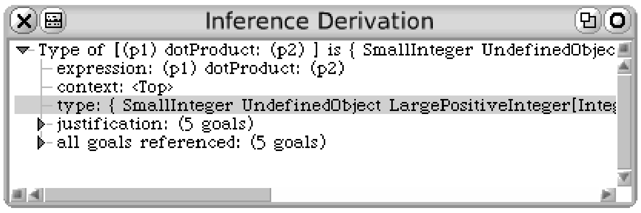
As mentioned previously, the **DCPA** algorithm by Wang and Smith partitions objects by which `new` statement allocates them [72]. A type-inference algorithm crafted by Oxhøj, Palsberg, and Schwartzbach also partitions objects by allocation site [51].

A large number of alias-analysis algorithms partition allocated objects using "allocation sites" [37]. An allocation site is typically an invocation of `new` or `malloc()` as in **DCPA**.

Plevyak and Chien describe an adaptive algorithm that often avoids using instantiation-point tags when they would not be able to refine the analysis [53]. 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 potentially superfluous analysis is mitigated, however, by the ability of algorithms in the **DDP** family 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, independently of whether or not their analyses could have been merged without loss of precision.

```
| 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 13.12: Data-polymorphism occurs in numeric array computations.



```
Type of [(p1) dotProduct: (p2) ] is { SmallInteger UndefinedObjec
  expression: (p1) dotProduct: (p2)
  context: <Top>
  type: { SmallInteger UndefinedObject LargePositiveInteger[Inte
  justification: (5 goals)
  all goals referenced: (5 goals)
```

Figure 13.13: The analyzer succeeds on the example in Figure 13.12.

```
| c a other |
c := ValueHolder_1 new.
a := c.
a contents: 'hello'.
other := ValueHolder_2 new.
other contents: 12345.

other contents.
c contents
```

Figure 13.14: The code from Figure 13.6 with source tags added to the two mentions of ValueHolder.

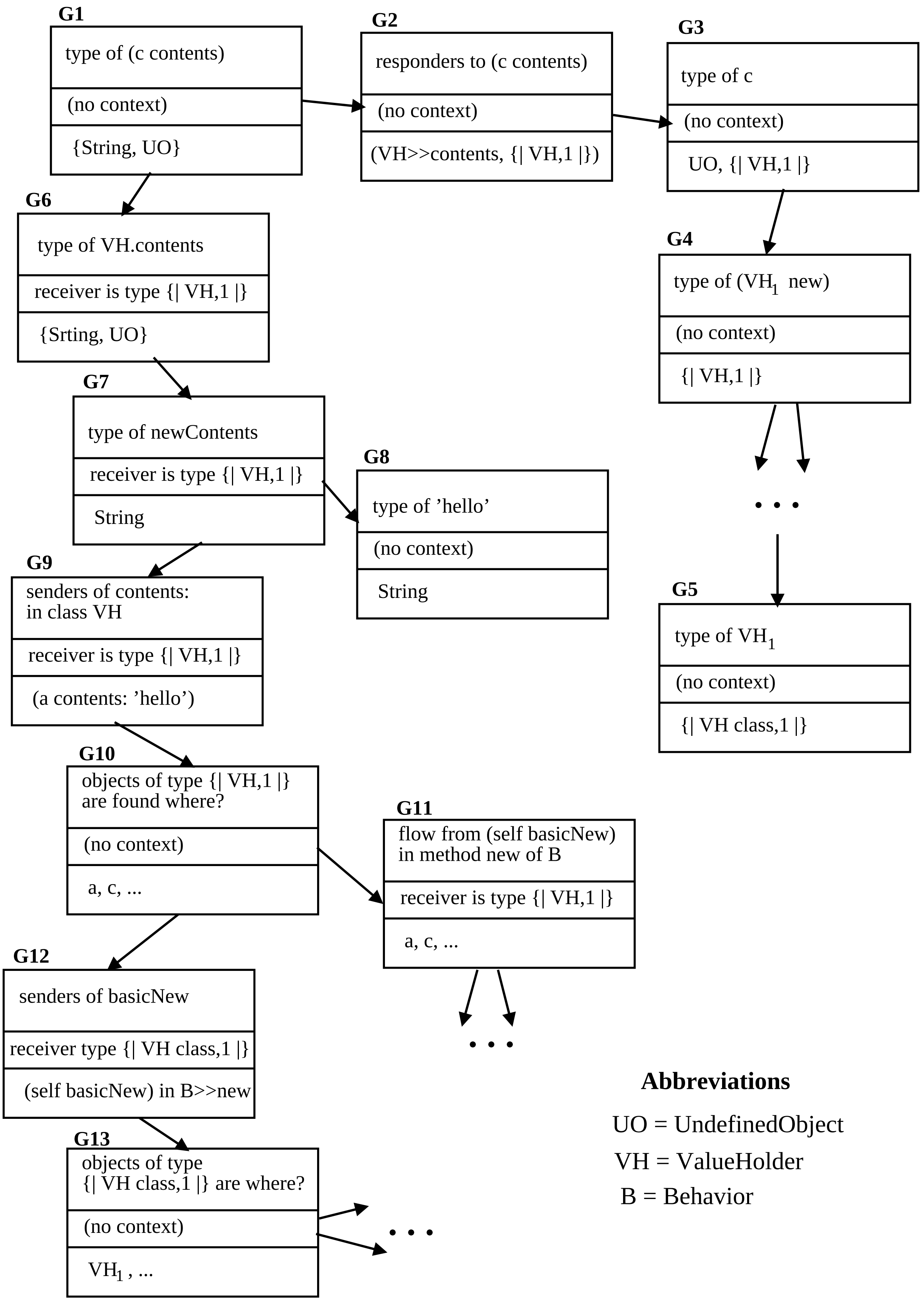


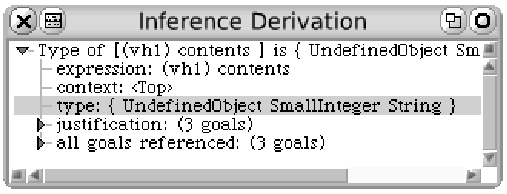
Figure 13.15: The graph of goals generated by **DDP/CT** when it infers a type for `c contents`, an expression in the method in Figure 13.14.

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

Figure 13.16: A typical factory method, `makeHolder`, for class `Platform`. This kind of indirection is useful to programmers in many circumstances, including the possibility that different platforms will implement the method to use a different value-holder class. Unfortunately for the analysis, however, all callers of this method will receive a `ValueHolder` with the same source tag: the single mention of `ValueHolder` in the `makeHolder` method.

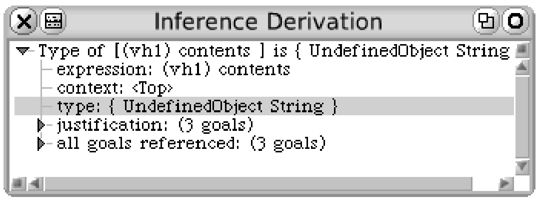
```
| vh1 vh2 |
vh1 := Platform makeHolder.
vh1 contents: 'hello'.

vh2 := Platform makeHolder.
vh2 contents: 123.

vh1 contents.
```

Figure 13.17: An example usage of the factory method from Figure 13.16. In this example, the inferencer as described so far fails to distinguish separate container objects, because both holders are given the same source tag.



```
Inference Derivation
▼ Type of [(vh1) contents ] is { UndefinedObject Sm
   expression: (vh1) contents
   context: <Top>
   type: { UndefinedObject SmallInteger String }
 ▶ justification: (3 goals)
 ▶ all goals referenced: (3 goals)
```

Figure 13.18: The analyzer merges flow through the two different holders in Figure 13.17, and so reports that `vh1` can hold both integers and strings.



```
Inference Derivation
▼ Type of [(vh1) contents ] is { UndefinedObject String
   expression: (vh1) contents
   context: <Top>
   type: { UndefinedObject String }
 ▶ justification: (3 goals)
 ▶ all goals referenced: (3 goals)
```

Figure 13.19: Using multi-level source tags on the example from Figure 13.17, it is possible to distinguish objects that are created via a factory object.

Chapter 14

Conclusions

This paper supports its thesis with the following work:

- a description of a new type inference algorithm to solve the stated problem
- a proof of correctness for this algorithm
- an implementation of the algorithm
- empirical analysis of the implementation's performance
- a complete application, Chuck, leveraging the implementation

The description shows that the algorithm meets the basic requirements in the thesis: the algorithm is demand-driven, it prunes subgoals, and it produces different types depending on calling context. The description also gives an argument that, intuitively, the algorithm should both scale and produce usefully precise types.

The proof shows that the algorithm infers correct types.

The implementations of the algorithm and the Chuck tool show that no pragmatic obstacles have been overlooked by the on-paper descriptions. The algorithm works in full Smalltalk and it generates the information needed for the main application intended.

The experimental results and the experiences with the Chuck tool both show that the algorithm finds usefully precise types in larger programs.

In addition to defending its thesis, my research makes the following contributions:

- It gives an operational semantics for the essence of Smalltalk. That semantics is thorough: it includes full closure semantics, nested mutable variables, and the `#perform :` method (`sendvar` in Mini-Smalltalk).
- It gives complete data flow rules for demand-driven analysis with CPA abstract contours for Smalltalk, including precise analysis of code using `#perform :` and blocks. These rules analyze forward flow in addition to type inference in order to support these features without being extremely conservative.
- It describes a general algorithm—demand-driven with subgoal pruning—that appears promising for other analysis problems. The general algorithm allows stronger inference rules to be used without abandoning scalability.
- It provides a complete implementation of the specific algorithm **DDP**.
- It provides an analysis framework for Smalltalk, used by the **DDP** implementation, that efficiently supports interactive programming.

- It provides a program understanding tool, Chuck, that takes advantage of the **DDP** implementation, thus bringing **DDP**'s advantages to practitioners.
- It empirically identifies effective pruning thresholds for **DDP**, so that future implementors of **DDP** have a good initial tuning of the algorithm's main parameter.
- It empirically identifies the most common types that appear in a representative sample of Smalltalk code.
- It empirically quantifies the improvement of subgoal pruning over root-goal pruning for **DDP**.

This work contributes to three major discussions that are ongoing in the programming language community.

First, it reopens the problem of context-sensitive type inference in larger programs. The prevailing research on type inference for the last ten years or so has reduced various kinds of sensitivity in order to achieve scalability. For example, researchers have removed directionality from the data flow, they have removed the use of precise call graphs, and they have merged goals for multiple expressions into just one. My work adds a new option that scales while remaining context-sensitive and while using directional data flow.

Second, my work emphasizes a connection between two existing fields: program analysis and knowledge-based systems. Demand-driven algorithms, in general, are actually simple knowledge-based systems where each goal has only one rule available for solving it. This connection between the fields seems likely to be fruitful. The encoding of many analysis algorithms as knowledge-based systems appears likely to be straightforward. Using a knowledge-based system as the overall architecture, lets a program analyzer attempt a variety of strong inference rules, without fully committing to the worst-case cost of those rules. This advantage is not specific to type inference.

Finally, this work continues a long-running discussion in programming language design regarding static and dynamic languages. Since **DDP** does infer precise types even in large programs, it seems that type inference is practical even when a language is not statically type-checked. Thus, this research defends language designs where one begins with a dynamic language and then adds type-based features as an option to be applied whenever and wherever a software engineer deems it most useful. In short, this research enriches the design space between static and dynamic languages.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

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

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

[4] Gagan Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *ICSM*, pages 453–462, 1999.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture (FPCA)*, pages 31–41, New York, NY, USA, 1993. ACM Press.

[7] Jonathan Aldrich and Criag Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[8] 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.

[9] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[10] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[11] Andrew J. Barnard. *From types to dataflow: code analysis for an OO language*. PhD thesis, Manchester University, 1993.

[12] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 133–141, 1982.

[13] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 213–223, New York, NY, USA, 2003. ACM Press.

[14] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1993.

[15] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[16] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[17] Craig Chambers. The cecil language specification and rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering, University of Washington, March 1993.

[18] Matthew J Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, University of Virginia, 1998.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[20] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 222–236. ACM Press, 1998.

[21] Danny Dubé and Marc Feeley. A demand-driven adaptive type analysis. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 84–97. ACM Press, 2002.

[22] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Symposium on Principles of Programming Languages*, pages 37–48, 1995.

[23] 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.

[24] Cormac Flanagan and Matthias Felleisen. A new way of debugging lisp programs. In *Proceedings of Lisp Users' Group Meeting (LUGM)*, 1998.

[25] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):370–416, 1999.

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

[27] Fransisco Garau. Inferencia de tipos concretos en Squeak. Master's thesis, Universidad de Buenos Aires, 2001.

[28] Jean-Yves Girard. *Interprétation Fonctionelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

[29] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

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

[31] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 136–150, New York, NY, USA, 1990. ACM Press.

[32] 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.

[33] Mark J. Guzdial and Kimberly M. Rose. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[34] Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science*, pages 342–351, 1997.

[35] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[36] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 448–472. Berlin: Springer-Verlag, 1991.

[37] 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, UT, 2001.

[38] 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.

[39] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 393–407, New York, NY, USA, 1995. ACM Press.

[40] Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. *Handbook of logic in computer science*, 4:527–636, 1995.

[41] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27(1):128–145, 1980.

[42] Alan C. Kay. The early history of Smalltalk. In *The second ACM SIGPLAN conference on History of programming languages*, pages 69–95. ACM Press, 1993.

[43] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[44] Donald E. Knuth. An empirical study of FORTRAN programs. Technical Report RC 3276, IBM Research, 1971.

[45] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, 1991.

[46] Xavier Leroy. Polymorphic typing of an algorithmic language. Research report 1778, INRIA, 1992.

[47] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), December 1978.

[48] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, August 1990.

[49] Chris Nikolopoulos. *Expert Systems: Introduction to First and Second Generation and Hybrid Knowledge Based Systems*. Marcel Dekker, Inc., New York, 1997.

[50] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459, May 1993.

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

[52] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[53] 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.

[54] Fran cois Pottier. A framework for type inference with subtyping. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 228–238, New York, NY, USA, 1998. ACM Press.

[55] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 163–196, 1993.

[56] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[57] 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.

[58] Mark A. Sheldon and David K. Gifford. Static dependent types for first class modules. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 20–29, New York, NY, USA, 1990. ACM Press.

[59] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[60] Olin Shivers. The semantics of Scheme control-flow analysis. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, SIGPLAN Notices, Vol. 26, No. 9, pages 190–198, Yale University, June 1991. ACM Press.

[61] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *Software Engineering*, 26(9):849–871, 2000.

[62] R. B. Smith and D. Ungar. Programming as an experience: The inspiration for self. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[63] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[64] Norihiss Suzuki. Inferring types in Smalltalk. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 187–199, 1981.

[65] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual: Language and Standard Libraries*. Springer, 1997.

[66] Aaron Melvin Tenenbaum. *Type determination for very high level languages.* PhD thesis, Courant Institute of Mathematical Sciences, 1974.

[67] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[68] 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.

[69] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[70] D. Ungar and R. B. Smith. Self: The power of simplicity. In *ACM Conference on Object-Oriented Programming, Systems, Language, and Applications (OOPSLA)*, 1987.

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

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

[73] Mario Wolczko. Semantics of Smalltalk-80. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 108–120, London, UK, 1987. Springer-Verlag.

[74] Mario Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, University of Manchester, 1988.

Index