Imperial College London
Department of Computing

Inferring Useful Static Types for Duck Typed Languages

Alexander Lamaison

September 2012

Supervised by Professor Alexander L. Wolf

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I hereby certify that all material in this thesis which is not my own work has been properly acknowledged.

Alexander Lamaison

Abstract

Complete and precise identification of types is essential to the effectiveness of programming aids such as refactoring or code completion. Existing approaches that target dynamically typed languages infer types using flow analysis, but flow analysis does not cope well with heavily used features such as heterogeneous containers and implicit interfaces.

Our solution makes the assumption that programs that are known to work do not encounter run-time type errors which allows us to derive extra type information from the way values are used, rather than simply where those values originate. This is in keeping with the "duck typing" philosophy of many dynamically typed languages.

The information we derive must be conservative, so we describe and formalise a technique to 'freeze' the duck type of a variable using the features, such as named methods, that are provably present on any run of the program. Development environments can use these sets of features to provide code-completion suggestions and API documentation, amongst other things. We show that these sets of features can be used to refine imprecise flow analysis results by using the frozen duck type to perform a structural type-cast.

We first formalise this for an idealised duck-typed language semantics and then show to what extent the technique would work for a real-world language, Python. We demonstrate its effectiveness by performing an analysis of several real-world Python programs which shows that we can infer the types of method-call receivers more precisely than can flow analysis alone.

To my supervisor, Professor Alexander L. Wolf, and to Professor Sophia Drossopoulou. For their support and friendship and for making me laugh.
To my friends, just for being.

Contents

1 Introduction

Users of statically typed languages are accustomed to development environments that ease the programming burden by suggesting, warning and partially automating the process. Although environments for dynamically typed languages try to provide these same features, they tend to fall short because techniques that work well with statically typed languages do not cope as well with the increased degrees of freedom allowed by dynamic typing. This freedom permits greater flexibility but reduces the assumptions we can make about the run-time behaviour of programs, assumptions upon which programmer assistance critically depend.

At the heart of the problem lies *static type inference*: how to conservatively approximate the run-time values of an expression without executing the program. Programming environments rely on these approximations in order to reason about a program and ideally they should be both precise, so as to help the user as much as possible, and sound, so as to not mislead the user. However, that is not possible because any tractable, sound, static analysis must be imprecise to some degree [25, 31]. The aim, then, is to make an analysis as precise as possible because the closer it approximates reality the better the quality of assistance.

In the absence of a static type system, dynamically typed language tools typically rely on *flow analyses* to infer types. As general program analyses, rather than program verifications, they impose no restrictions on program behaviour, which means they work in a dynamically typed setting [41]. However, they are more effective with statically typed languages because the *interface types* from the static type system limit the effect of any imprecise *concrete types* inferred by the flow analysis [42].

The question becomes: is it possible to use interface types to achieve the same effect in the absence of a static type system? In this thesis we show that, subject to some caveats, it is possible for *duck typed* languages.

1.1 Duck typing

Duck typed languages, which include Python, Ruby and Smalltalk among many others, are dynamically typed languages characterised by their run-time type checks, which test a value's capabilities instead of its identity—the so called *duck test*: "If it looks like a duck and quacks like a duck, it must be a duck".[1] There are no accepted definitions of duck typing, so for the remainder of the thesis we assume the following:

Duck typing A dynamic, lazy form of structural typing. Dynamic because safety checks occur at run time. Structural because a value's compatibility is decided solely based on its interface. Lazy because compatibility is checked as late in the execution as possible using as little of the interface as possible.

Although duck types are interface types, they are dynamic, not static. The types do not exist until run time and are not valid beyond the single execution that emitted the series of checks. Fortunately, unlike concrete types, interface types need not correspond to actual values, so we can obtain a sound static approximation of a program's duck types by factoring out only those parts of the interface that will definitely be checked on any run of the program.

Freezing ducks Our first contribution is a two-part static analysis for duck typed languages that extracts a static subset of an expression's duck type. The interface types it produces are sets of *features*, such as method calls, that the values at each expression must possess if the program is to pass all run time duck tests. Notably, the analysis is *flow sensitive* (section 2.5.4) meaning it infers a separate type for each expression, which is essential for a dynamically typed language where the programmer is free to coordinate polymorphism themselves (section 1.3).

The first part of the analysis, inspired by the work of Adams et al. [1] for Scheme, collects *observed features* of an expression: features that are duck-tested for, and found to be present, in every value that flows to the expression before it reaches the expression. Given an assumption that values have static interfaces and that failing duck tests halt execution, these

[1] `http://docs.python.org/glossary.html#term-duck-typing`.

features are guaranteed to be present in any value at that expression, under all circumstances. The analysis is sound.

The second part of the analysis is a novel application of success types, originally for Erlang by Lindahl and Sagonas [23]. It collects *required features* of an expression: features that are duck tested for, but not necessarily found to be present, in every value that flows to the expression after it reaches the expression. Given the same assumption of static interfaces, these features are guaranteed to be present in any value at that expression if the program never encounters a failing duck test. The analysis is sound for a *well formed* program.

Using both parts of the analysis together renders the most precise interface type, but certain applications that must reason about the behaviour of potentially ill-formed programs, such as type checkers and compilers, should omit required features from the frozen duck type as the result is not sound under those circumstances. However, for our application—programmer assistance—we assume the input program is well formed and use both parts of the analysis.

1.2 Contraindication

Our second contribution is *contraindication*: an approach that combines the frozen duck types with the concrete types from flow analyses by using the former to down cast the latter, just as the interface types from a static type system do for flow analysis in a statically typed language. The result is a more precise type than is possible using either frozen duck types or flow analysis in isolation.

1.3 Coordinating polymorphism

To illustrate the problem, let us consider an example in the duck typed language Python that, although well formed, takes advantage of dynamic typing in a way that makes a precise concrete type uncomputable by flow analysis.

```
class A:
    def wibble(self): ...
class B:
 4      def foo(self): ...
 5  class C:
 6      def foo(self): ...
 7      def bar(self): ...
 8
 9  if x.feminine():
10      p = A()
11  else if x.masculine():
12      p = B()
13  else:
14      p = C()
15
16  if x.neuter():
17      e = p.foo()
18      f = p.bar()
```

The challenge with dynamically typed code like this is that static analysis is unable to precisely predict the run-time paths through the code; it must *over*-approximate the possibilities and must then attempt to reason about situations that never occur in practice. At best this results in imprecision in the inferred type, but at worst the situation becomes fatally conflicted and the analysis must abandon inference and return a trivially pessimistic answer. Our results have shown this worst-case scenario happens almost half the time for a context-insensitive flow analysis (chapter 7); it is not an edge case.

In this example, the type of the object in p is the stumbling block. It depends on the run-time value of x and so can contain instances of classes A, B or C. However, at lines 17 and 18, p can only contain an instance of class C as no path that passes through lines 10 or 12 can take the true branch of the conditional at line 16. This fact is obvious to the programmer and something they, presumably, relied on when writing this code, but a flow analysis would need to establish

$$\mathbf{not}(\mathsf{x.feminine()}\ \mathbf{or}\ \mathsf{x.masculine()}) \equiv \mathsf{x.neuter()}$$

for all runs of the program, which, in the general case where results may even depend on input, is uncomputable.

The issue stems from the nature of dynamically typed languages that permit the programmer to freely *coordinate polymorphism* along run-time paths using arbitrary data and control structures. A static type system, on the other hand, tightly regulates all polymorphism such that even the statically over-approximated paths are provably type safe. Any deviations from the type system's regulations must be forced by the programmer through the use of casts, which may fail at run time. A side effect of type systems is the rejection of a subset of well-formed programs [31], including this example.

Development tools for dynamically typed programs cannot afford the luxury of being able to reject existing working programs. We must use techniques that accept "arbitrarily objectionable" [41] programs.

1.4 Flow analysis

Flow analyses use data-flow information to conservatively approximate the sources of the values that can reach an expression [27, 42] without the restrictions of a static type system. As general program analyses, rather than verifications, flow analyses can be used to infer types for all programs, even those whose type safety cannot be verified. This makes them a common basis for type inference in development tools for dynamically typed languages.

The success of a flow analysis relies on being able to trace data-flow paths between an expression and *all* the sites where its values may have been created, namely constructors, constants or allocations. As program size increases so does the potential for imprecision to obscure this data-flow connection. Nevertheless, flow analyses are sound and, as such, err on the side of caution and infer types that over-approximate values in a program.

A flow analysis might infer the set of possible types of p at lines 17 and 18, denoted $[\![\mathsf{p}_{17}]\!]$ and $[\![\mathsf{p}_{18}]\!]$, as $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. This type is conservative but it is not precise.

While the loss of precision in the inferred type of p is inconvenient, the knock-on effect for $[\![\mathsf{e}_{17}]\!]$ and $[\![\mathsf{f}_{18}]\!]$ is truly fatal. The type of values that arrive in e and f depend on the return types of the calls to $\mathsf{p.foo()}$ and $\mathsf{p.bar()}$ that, in turn, depend on $[\![\mathsf{p}_{17}]\!]$ and $[\![\mathsf{p}_{18}]\!]$. But $[\![\mathsf{p}_{17}]\!]$ and $[\![\mathsf{p}_{18}]\!]$ contain classes that do not have methods foo or bar so, the flow analysis, being a conservative analysis, must return the trivially pessimistic result often called $\top$ and pronounced 'top'.

1.5 Feature analysis

Even when flow analysis fails, there is other information an analysis can consider to regain some precision: information about how a value is used rather than where it came from.

Firstly, a value reaching an expression must have passed any duck tests to which it was subjected. The first part of our feature analysis uses the duck tests, which it can guarantee will always happen, to infer a set of *observed features* for the expression. In the example, any value reaching p_{18} will already have been tested for the presence of method foo when the value passed through line 17, so having a method foo is a feature of p_{18}. The important point is that it is not just *possible* that the value appearing in p_{18} will have passed a check for foo before it arrives there, it is *guaranteed.* No matter how freely the user has coordinated polymorphism in their program, the features inferred by the first part of the analysis will always be present because only guaranteed checks are considered. The analysis is sound.

But values in any sensible program must also pass the duck tests to which they are subjected *after* arriving at an expression. The second part of our feature analysis considers duck tests that inevitably will be executed on any value that reaches the expression: the set of *required features* for the expression. In the example, a value reaching p_{17} will be tested later for the presence of method bar when it passes through line 18. This time it is not guaranteed that the values appearing in p_{17} will have feature bar; the duck test on line 18 could still fail. However, assuming the user coordinated their polymorphism correctly—the program is well formed and never fails a duck test—the inferred features are once again guaranteed, and the analysis is sound.

This last assumption is unusual and sets us apart from most work on static typing, which is aimed at type checking, type safety, compiler optimisation and other applications that must accept ill-formed programs. However, we argue that, in our domain of programmer assistance, an assumption that the code is well-formed is reasonable, standard and fruitful.

Reasonable because a project cannot reach even moderate maturity if still encountering run-time type errors. Even during development, when some code may not be correct or complete, the majority of the system will still be type correct. The quality of the assistance available should not have to be compromised for the sake of small sections of new and untested code.

Standard because even in statically typed languages, development environments do not promise correct assistance if the code does not pass the type checker. Refactoring engines, for example, refuse to operate entirely under those circumstances. The difference here is simply that we do not have a type checker to warn the user of the problem beforehand, a problem that is not caused by nor exacerbated by our assumption.

Fruitful because our results show the clear benefits of using required-feature analysis over an analysis that considers observed features alone (chapter 7).

Together, the observed and required features inferred by the two analyses form an expression's *frozen duck type.* These are already useful in their own right. For example, a development environment can present the user with a set of features for an expression, and the user can rely on those features always being available in the values at that expression. Or an environment can reverse-engineer API documentation from the source. The tool can document (perhaps interactively) that an argument being passed to an API must have at least the inferred required features.

But our main aim is to use these types to recover the casts that would have been present in a statically typed language and use them to refine the results of a flow analysis.

1.6 Recovering casts

The concrete types inferred through flow analysis are *nominal* whereas the frozen duck types are structural interface types. But the two different notions of type can be combined when the nominal type is a set of concrete types whose definitions constrain the set of features their values support. For example, where values are created through named value-classes and those classes declare all the features the value will ever support, the nominal and structural types are related by the sets of features they contain.

Looking at the example again, we can use *observed features*—the first part of our duck-type analysis—to reduce the nominal type $[\![\mathsf{p}_{18}]\!]$ to the subset of classes inferred by flow analysis that mandate the presence of all the observed features of p_{18}, in this case just method foo. The inferred concrete type is down cast to the set of classes that all define a method foo. In this case:

$$\begin{aligned}
\llbracket \mathsf{p}_{18} \rrbracket &= \underbrace{\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}}_{\text{from flow analysis}} \cap \overbrace{\{\tau \mid \tau \in \top \wedge \mathsf{foo} \in \text{features}(\tau)\}}^{\text{from observed features}} \\
&= \{\mathsf{B}, \mathsf{C}\}
\end{aligned}$$

where features(τ) is the set of methods defined in class τ and $\top$ is the set of all possible classes.

If we now include *required features* as well, we can down cast the inferred type further as long as we are happy to assume the program does not encounter a failed duck test. If our example is a well-formed program, the value of p_{18} must not only have a method foo but also a method bar, given the call to bar at line 18. Thus, any type not defining bar can also be eliminated from $\llbracket \mathsf{p}_{18} \rrbracket$.

$$\begin{aligned}
\llbracket \mathsf{p}_{18} \rrbracket = &\underbrace{\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}}_{\text{from flow analysis}} \cap \overbrace{\{\tau \mid \tau \in \top \wedge \mathsf{foo} \in \text{features}(\tau)\}}^{\text{from observed features}} \\
&\cap \overbrace{\{\tau \mid \tau \in \top \wedge \mathsf{bar} \in \text{features}(\tau)\}}^{\text{from required features}} \\
= &\underbrace{\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}}_{\text{concrete type}} \cap \overbrace{\{\tau \mid \tau \in \top \wedge \{\mathsf{foo}, \mathsf{bar}\} \subseteq \text{features}(\tau)\}}^{\text{interface type}} \\
= &\{\mathsf{C}\}
\end{aligned}$$

The analysis for $\llbracket \mathsf{p}_{17} \rrbracket$ would proceed similarly except that both foo and bar would be required features of p_{17} and there would be no observed features.

Imagining, for a moment, that the source code could be modified to make our casts explicit, the frozen duck types might cast the types structurally like so:

```
16  if x.neuter():
17      ((:foo, :bar) p).foo()
18      ((:foo, :bar) p).bar()
```

which, when combined with the concrete type through contraindication, might become:

```
if x.neuter():
    ((C) p).foo()
    ((C) p).bar()
```

1.6.1 Soundness

Contraindication constrains the *concrete types* inferred by flow analysis using the *interface types* of our frozen duck types. Concrete types produced by flow analysis over-approximate the actual values at an expression while our frozen duck types, being interface types, under-approximate the actual interfaces of the values. By contraindicating only those members of the concrete type whose interface is incompatible with the interface type, we arrive at a result that is still sound.

1.7 Contributions

- We adapt the work of Adams et al. [1] to suit a duck typed language semantics (section 3.5) and generalise their approach in our *observed features* to permit non-halting definitions of run-time type errors (section 3.7).
- We present an analysis to approximate observed features based on standard intra-procedural dominator and kill analyses (section 3.10).
- We adapt observed features in the spirit of success typings [23] to become *required features*, which consider operations due to occur rather than those that will already have happened (section 3.8).
- We present an analysis to approximate required features based on standard dominator and kill analyses (section 3.10).
- We present *contraindication*, a technique to refine concrete type inference by down casting the inferred types using structural interface types in a class-based language (section 3.6).
- We present evidence of the improved type precision by applying contraindication to a range of real-world programs (chapter 7).

Outline

Chapter 2 Related work: We explore the existing work in the field, both work that we build upon and work whose goals coincide with ours but whose approach is separate.

Chapter 3 Approach: We develop the theoretical foundations of our approach by combining aspects of three different pieces of work, adapting them, when necessary, to better suit a duck typed language semantics. We outline the basis of a practical approach to calculate a conservative approximation of frozen duck types and explain how to use them for contraindication in a class-based language.

Chapter 4 Formal presentation: We formalise observed and required features as well as contraindication. We prove that these are sound and define the necessary prerequisites that make them so. They are, however, uncomputable, so we strengthen the definitions in stages to result in an approximation that is computable using an intra-procedural control flow graph. We prove that this approximation is sound.

Chapter 5 Practical language: We explore the challenges and compromises of applying the approach to a practical language by mapping the concepts to the semantics of Python.

Chapter 6 Implementation: We describe the implementation of our approach that we use to produce the results in chapter 7. This implementation includes a context-insensitive flow analysis whose results we refine.

Chapter 7 Evaluation: We study the effect of contraindication on a set of open-source Python programs. As well as presenting the improvements in precision, we verify our result by manual inspection of a sample. The chapter ends with a discussion of the threats to the validity of this study.

Chapter 8 Conclusion: We conclude with a discussion of open problems, new avenues of research and further work we hope to do. In particular, we discuss our desire to integrate the analyses into an IDE for Python.

2 Related work

In this chapter we explore the existing work in the field. We begin with two categories of work that address the same issues as this thesis but whose approaches are notably different. The first relies on users adding type annotations to assist static analysis, while the second captures actual types from running programs. We aim to infer types statically without needing the user to add type annotations, so in the rest of the chapter we explore the work that confronts that particular challenge.

We consider type systems designed to ensure type safety, that reject some working programs, as well as those that incorporate run-time type checks into the system and so accept all programs. Then we describe a family of general program analyses commonly used for type inference in dynamically typed languages and for optimisation in statically typed languages. We demonstrate some of the weaknesses of using this kind of analysis with dynamically typed languages and explain why statically typed languages are less vulnerable. Finally, we discuss work that recovers some of these advantages by exploiting other sources of type information in dynamically typed programs. The approach we develop in this thesis builds on this work by adapting it to duck typed languages, a popular class of dynamically typed languages, extending the sources of information used to improve precision and then proving that the approach yields sound types for well-formed programs.

2.1 Annotations

Some type inference approaches for dynamically typed languages require the user to add type annotations to their source code. Although type annotations are common in statically typed languages, such as C++ or Java where every variable and parameter must have its type declared the first time it appears, most dynamically typed languages do not require them.

We believe that programmers will not be willing to annotate existing code bases, especially as our type inference is meant to reduce, not increase, their workload. Therefore we discuss the work here for the sake of completeness but we do not base our approach on it. Later we will cover work that infers types from unmodified dynamically typed code.

2.1.1 Gradual typing

As the name implies, gradual typing [38, 39] eases the transition from untyped to typed programming by letting the user mix annotated and unannotated code. Gradual typing reconciles the dynamically typed and statically typed parts of the program by inserting checked casts at the interface between statically and dynamically typed portions of the code.

Gradual typing is similar to soft typing (section 2.3.2) but only statically type checks code that has type annotations and treats unannotated code dynamically. When a value originates in dynamically typed code, a run time check occurs before it is used in a statically typed context [49]. Soft typing, on the other hand, attempts to statically type check all code and only inserts a run time type check if that fails.

DRuby [14] is a type checker that applies gradual typing to a duck-typed language. As well as statically checking annotations, DRuby attempts to infer the types of unannotated code using the extra information the annotations provide.

2.1.2 Like types

Like types are a recent development introduced in the language Thorn [49]. They improve on gradual typing by making it explicit when a variable could be wrapping a dynamically typed value that, therefore, might fail a runtime type check. This intermediate type integrates the statically typed and dynamically typed code without the former losing its robustness and performance advantages nor the latter losing its flexibility.

Like-typed variables have the same behaviour as the statically annotated code in a gradual type system: any value can be assigned to them as though they were dynamically typed while their uses are both statically and dynamically checked. Like types are structural, so the uses are checked to make sure they are compatible with the like type's interface alone; missing features cause a static type error. And, as with gradual typing, those statically checked uses can still fail at run time if the value in the variable, which is always assigned dynamically, does not support the features in question.

The advantage over gradual typing is that like types are a third, distinct, category of type [49] which allows static types to continue to work exactly as normal with all the usual advantages such as guaranteed type safety and optimisation.

Wrigstad et al. present like types in the context of a new language, but retrofitting them to an existing dynamically typed language would require modifications to the language and for the user to annotate any existing code they wanted to benefit from the like type.

When comparing like types with soft typing (section 2.3.2) the latter can be thought of as inferring where the like type annotations must be inserted. The trade off is that explicit like types allow for unequivocal type errors, whereas soft typing only produces warnings and does so every time a dynamic check is used, limiting its practical usefulness.

2.2 Run-time analysis

A very different way of obtaining type information is to do it at run time when it is, arguably, most natural for a dynamically typed language [18]. Furr et al. [14] augment their DRuby static analysis (section 2.1.1) with run-time profiles taken from instrumented executions of the test suite. Haupt et al. [18] harvest types by executing the test suite one instruction at a time and inspecting the state of the interpreter in between.

Such dynamic analyses are a valid and effective solution to the problem of providing better programmer assistance. They are even able to assist in the face of pathologically dynamic features such as `eval` and other features that execute arbitrary data as code. But such analyses perhaps best complement static type inference rather than replace it. In particular, they rely on the presence of a test suite to exhaustively exercise the code. If a particular value appears at an expression during a real execution, but did not appear during testing, the collected types will be incorrect.

2.2.1 RPython

A notably different use of run-time type analysis is RPython [4] an intermediate language that forms part of the tool chain used to implement PyPy,[1] the Python implementation of Python. It is a restricted subset of Python that is entirely statically typed.

The interesting aspect is that it takes as input "live Python objects" from a running instance of the Python interpreter during an initialisation phase. After that point, all type information is available, so the initialisation phase is allowed to use all the dynamically typed features Python offers.

This approach is unsuitable for our domain of general program analysis, as we cannot restrict the dynamic behaviour of the input program to an initialisation phase.

2.3 Type systems

Type systems are typically used to guarantee the absence of certain behaviours statically, both for languages with explicit type annotations and, through type inference, for languages without [4,6,10,27,28,31,32,35]. The latter category are most interesting for us, as dynamically typed languages also, typically, lack type annotations.

2.3.1 Constraint-based type inference

Type constraints are a basis for type inference that record the dependencies between types of terms that are then solved to infer a type for the terms using a *unification* algorithm [31]. One of the best known unification algorithms is algorithm W by Milner for ML [12], which is efficient and, if a given term is typable, always infers the most general type for it. However, unification requires that there be a single type for all appearances of a variable, so would be unable to infer a type for `p` in our example from chapter 1.

This is not surprising, as static type checking inevitably excludes some programs that would not exhibit the excluded behaviour at run time, simply because the checker could not generate a proof [31]. When statically

[1]http://pypy.org/

inferring types for a dynamically typed language, we do not have the luxury of rejecting some working programs, since the technique must accept all working programs if it is to be faithful to the semantics of the language.

However, some work on type systems has tried to address the issue by incorporating run time checks into the type system.

2.3.2 Soft typing

Cartwright and Fagan introduced soft typing [11], an approach to type systems designed to cope with dynamically typed programs by inserting run-time checks when a program fails to statically type check. Unlike a traditional type system, a soft typing system never rejects a program if it fails to infer a static typing for it. Instead, it inserts "narrowers" in the code to "transform arbitrary programs to equivalent programs that type check" [11] when necessary.

Narrowers are type casts and indicate where run-time checks must be inserted into the program. The effect of the narrower is to resolve contradictions encountered by the inference algorithm by blindly converting the source type to the destination type and relying on the run-time checks to catch situations where the types are truly incompatible. Exactly how this narrowing is done depends on the particular static type system upon which the soft typing is built. Cartwright and Fagan [11] use an algorithm similar to algorithm W [12], so the contradictions are failures to unify a set of type constraints.

Soft typing is similar to gradual typing [38, 39] (section 2.1.1): both reconcile dynamically typed and statically typed parts of the program by inserting checked casts where necessary. But soft typing tries to infer static types for as much of the program as necessary in the absence of any type annotations. If it fails to do so, it assumes the code is dynamically typed, with the consequence that it cannot detect type errors.

There are similarities between our work and soft typing. Both techniques reason about untypable code. Our analysis assumes the code is type correct and works out how that came about. Soft typing accepts that the code might not be type correct and works out how to check that it is. Both techniques down cast types, but they do so with very different aims in mind.

Soft typing has two main aims: to optimise a program by reducing the number of run-time type checks and to warn the programmer of potential type errors [11]. Both run-time type checks and potential type errors occur precisely where casts are inserted in the type system, so the fewer casts a soft typing algorithm inserts the faster and more type safe a program will be.

Our approach, contraindication, refines imprecise types by down casting them, such that they become consistent with the operations being performed: the type resulting from the cast would have made the program pass a type check.

2.3.3 Hybrid typing

A variation on soft typing, hybrid typing, restores the ability for the type system to type check a program by dividing expressions into three classes: those that are statically type correct, those that are statically type incorrect and those that must be dynamically checked. The dynamically checked ones can continue to be warnings for the programmer, but the statically type-incorrect category are unequivocally type errors.

2.4 Success typings

Success typings are a very different kind of type, introduced by Lindahl and Sagonas [23] in the context of Erlang, a dynamically typed, functional language. Unlike typical static types, which guarantee type safety and therefore are uncomputable for a dynamically typed language, success types guarantee that any use contradicting them will definitely lead to a run-time type error. They are particularly suited to dynamically typed languages as they require no type annotations and are always sound for any well-formed program. In particular, they will never reject a well-formed program but might accept ill-formed programs, making them suitable for type checking but not type safety.

Our *required features* (see sections 1.5, 3.8 and 4.5) are similar to success typings but, unlike the Lindahl and Sagonas approach, we do not (currently) use them for type checking. Instead, we assume a well-formed program that will not encounter a run-time type error and use them to detect where the flow analysis inferred too wide a type. The way we calculate our required features is very different from Lindahl and Sagonas as they operate in the context of an Erlang-style purely functional programming language with immutable variables, while we generalise our approach to include imperative languages with reference semantics and mutable variables.

2.5 Flow analysis

Flow analyses are a family of general program analyses with a long history in computing [3]. They have been used for compiler optimisation [8, 36] (call graph construction [16, 46], closure conversion [44], alias analysis [5, 45]), safety checking [28] and type inference [13]. They have been used for procedural [5, 45], functional [36] and object oriented languages [26, 28, 32, 46], both statically [13, 16, 48] and dynamically typed [8, 26, 28]. There are a wide variety of implementation strategies, but they all share the same basic idea that properties of a program can be predicted statically by *over-estimating* the *data flow* in a program.

The analyses model data flow in a program by simulating its execution, using *abstract* values to approximate program state. They either start with an assumption that all values can flow everywhere and apply rules that reduce this by proving a flow impossible or they assume nothing flows anywhere and apply rules that provide evidence that a flow is possible [15, 16].

Somewhat confusingly, despite being data-flow analysis, they are often called *CFAs* (control flow analyses) due to the circular nature of data flow and control flow in higher order languages. The acronym is often used with a prefix to identify a specific variant (see section 2.5.1) of which there are many.

2.5.1 Variants

Like all static analyses, flow analyses, at best, approximate actual program behaviour. As flow analyses are so generally useful, a large amount of work has been done to make the approximation as precise as possible.

There are two main flavours of implementation.

Propagation Also called Andersen's analysis [5], dependencies between expressions are *inclusion* constraints, which accurately model the direction of data flow.

Unification Also called Steensgaard's analysis [45], dependencies between expressions are *equality* constraints, which ignore the direction of data flow.

Sets of equalities are easier to solve than sets of inclusions [22, 25], but the result is less precise because the constraints no longer model the actual data flow as accurately.

All analyses inevitably make compromises because no static analysis can precisely model an arbitrary program's true behaviour. When the analyses are used for type inference, the approximations typically manifest as imprecise modelling of polymorphism. Different compromises affect the way the analyses approximate different aspects of a program's behaviour and in the rest of this section we discuss some approximations (characterised, as usual in the literature, by their complement: what the analyses are *sensitive* to) and demonstrate how they manifest themselves as over-approximation in the modelling of polymorphism.

The simplest type of flow analyses are known as *monovariant* because they maintain one abstract value per syntactic element [43]; some analyses [46] maintain even fewer. When the analysis is being used for type inference, this means that everywhere the abstract value might flow must share the same type, which must over-approximate the actual types of *all* the actual values represented by that abstract value. The result can be fairly catastrophic because very general types flood the analysis. We describe this in greater detail in section 2.5.6.

The effect is not so severe in statically typed languages as the existing static types from the type system can be used to bound the imprecision [41] (see section 2.6) but it makes monovariant type inference less suitable for dynamically typed languages.

The syntactic elements are representations of memory locations and, depending on the programming language, may include parameters, fields and mutable variables. Each one may be polymorphic, but rarely arbitrarily so. Blindly summarising the polymorphism with a single abstract variable exaggerates the polymorphism and reduces the precision of the result.

Now we look at each of these syntactic elements, how summarisation exaggerates their polymorphism and what can be done about it. Parameters (parametric polymorphism) are dealt with by context sensitivity (section 2.5.2), fields (data polymorphism) by object sensitivity (section 2.5.3) and variables by flow sensitivity (section 2.5.4). In section 2.6 we show how static type systems bound the problem, making it less apparent in statically typed languages.

2.5.2 Context sensitivity

Procedure calls act as natural summarisation points in a flow analysis. Programmers call a procedure from more than one place in order to share some behaviour between the call sites. Context insensitive analyses reflect this shared behaviour by analysing each procedure only once regardless of the call sites. The result of a function is modelled by a single shared abstract value, likewise for variables within a procedure.

But, although procedures may share behaviour, that does not imply the behaviour is identical on every call. Maintaining only one set of abstract values per procedure leaves a context insensitive analysis unable to model these differences. When used for type inference, the result is that all calls to a function will always be given the same type, as will all parameters in a procedure.

The simplest example of the problem is the identity function, shown here as f:

$$\begin{array}{l}\mathsf{let}\ f = \lambda x.x\ \mathsf{in}\\ \mathsf{let}\ g = \lambda y.f\ y\ \mathsf{in}\\ \mathsf{let}\ h = \lambda z.\lambda w.z\ \mathsf{in}\\ f\ g\ h\end{array}$$

Here two different values flow to x, the parameter of f, first the abstraction g, which has type $\alpha \to \alpha$, then abstraction h, which has type $\alpha \to (\beta \to \alpha)$, via the body of g. A context insensitive analysis would combine these and assign x in f the type $\alpha \to \alpha \cup \alpha \to (\beta \to \alpha)$ and the imprecision would flow through the program, first to the inferred type of f—now $\alpha \to \alpha \cup (\alpha \to (\beta \to \alpha) \to (\alpha \to \alpha \cup \alpha \to (\beta \to \alpha))$ rather than $\alpha \to \alpha$ and then to the type of g.

Alternatively, in an object-oriented language like Python:

```
class A:
    ...
class B:
    ...

def identity(x):
    return x

a = A()
y = identity(a)
b = B()
z = identity(b)
```

a context insensitive analysis uses the procedure body as the unit of summarisation and would infer that, while a and b were monomorphic, y and z are polymorphic and may be instances of A or B. The problem is that the parametric polymorphism of parameter x has been exaggerated by the summarisation of all calls to identity. Across call sites, x is polymorphic but at an individual call site it is actually monomorphic. Notice how the values went into the function more precise than they came out. Below we describe a family of context sensitive analyses (k-CFA) [36] that model monomorphic call sites for a polymorphic procedure and, up to a certain call depth (k), even monomorphic calls at a polymorphic call site. Another analysis, CPA [2], can do the latter for all call sites regardless of depth.

The effect can be particularly acute if the function is a commonly used system function and a variation of the identity function, for example a filter function. The output type at *any* call site becomes polluted with the input type at *every* call site.

Static types improve the situation because the type checker will have assigned an interface type to the result of the identity function that is sufficiently precise to make the program well typed. In an explicitly-typed language such as Java, this may require the user to insert explicit casts that provide the extra type information (see figure 2.1 on page 34). We discuss this in more detail in section 2.6.

Context sensitive analyses regain some precision by distinguishing different calls to the same procedure by an approximation of their calling context, called a contour [15]. Calls to the same procedure with different contours are given distinct abstract values whilst calls with matching contours share abstract values.

Exactly what form this context takes varies between analyses. Ryder [34] and Grove et al. [16] include detailed surveys of the variations. Below we cover two kinds of context, but these are by no means mutually exclusive.

Call strings: Call string contours are a representation of the call stack leading to the call site [15,16,34]. A true representation of the call stack may be infinite, so the representation is approximated by limiting the number of levels it models. A well-known family of context sensitive analyses, k-CFA by Shivers [36], maintains k levels of contour. Maintaining this separation is still hugely expensive [16, 46] and they do not scale for any k $>$ 0 [37]. 0-CFA is the monovariant (context insensitive) member of the family.

Parameter state: These analyses use some encoding of the state of the values passed to procedure at a call site [15]. In an object-oriented language this can include the state of the receiver, which is passed implicitly to the procedure.

A notable analysis in this category is Agesen's Cartesian Product Algorithm (CPA) [2] which maintains a separate contour for every combination of arguments that are passed at a call site. It avoids redundant analysis inherent in the call strings approach because different call sites that pass arguments of the same type—the common case—share a contour. Precision is also better than the k-CFA family because every procedure body is analysed separately with monomorphic parameters; a call site that remains polymorphic after k-level expansion causes the k-CFA analyses to analyse the receiver body with polymorphic (i.e., imprecise) parameters [2].

2.5.3 Object sensitivity

As well as polymorphic parameters, languages with records—notably the object-oriented languages—can have polymorphic fields [31]. This is known as *data polymorphism* [26,43,48]. Basic analyses summarise all instances of a class with a single set of abstract values [34,48] and a corresponding loss of precision. Oxhøj et al. [26] found such an analysis [28] for the duck-typed, object-oriented language Smalltalk to be "next to useless" [26] in practice.

For example:

```
class Bus:
    ...
class Bicycle:
    ...

class Person:
    def __init__(self, transport):
        self.transport = transport

red_bus = Bus()
jack = Person(red_bus)
jacks_bus = jack.transport

shiny_bike = Bicycle()
jill = Person(shiny_bike)
jills_bike = jill.transport
```

The transport field of the Person class is polymorphic, but not within an object instance. However, an object-insensitive algorithm uses the class as the unit of summarisation and will infer that *both* jacks_bus and jills_bike have the same type {Bus, Bicycle}. As with the example of parametric polymorphism (section 2.5.2), this causes the data polymorphism to be exaggerated. Notice how the modes of transport went in more precise than they came out. Summarising the fields by class meant that the analysis lost the ability to reason precisely about the field instances and, instead, mixed the polymorphic possibilities arbitrarily when, in fact, only if Jack had chosen to ride his transport polymorphically should the full combination of possibilities apply.

The effect is particularly acute for heterogeneous collections, which are effectively instances of a class with an unbounded number of numerically named fields (see section 2.5.6).

More advanced (and expensive) analyses [26, 48], which we dub *object sensitive* here, distinguish different allocation sites (constructor calls) of the same class and maintain separate sets of abstract values for each. Spoon and Shivers [43] achieve a similar effect by tagging constructor calls with the call site and relying on parametric polymorphism (see section 2.5.2) to disambiguate them. As usual, the increased sensitivity has an associated cost: a worst-case quadratic increase in the case of Oxhøj et al. [26], for example.

Statically typed languages can achieve the same effect without resorting to an object-sensitive analysis by relying on static types from the type system. In Java, for instance, these would either come from casts (figure 2.2 on page 35) or from generics that make the data polymorphism explicit.

2.5.4 Flow sensitivity

A flow insensitive analysis takes no account of the order of execution [25]. Typically this means that the results for all appearances of the same variable are combined. This is the norm for flow analyses used for type inference in statically typed language. Even in Whiley [30] which does flow-sensitive structural typing, the flow sensitivity is limited to branches controlled by explicit type checks; other situations where arbitrary conditionals coordinate the polymorphism are not reflected in the type system.

Ryder [34] suggests that the impact of flow sensitivity is minimal, at least for object-oriented code, as methods tend to be small, and that the scalability problems have led to context sensitivity being favoured.

While this may be the case, we believe duck typed languages benefit more than statically typed languages because implicit interfaces that vary polymorphically along different paths through the same function are not uncommon, and these language also permit variables to be reused for unrelated types within the same function.

Analyses can combine aspects of these (and other) dimensions arbitrarily [15,16] to increase precision at the cost of performance. The combination providing best precision for a certain cost will vary by problem domain, language, programming style and code complexity [16].

2.5.5 Demand driven

Demand-driven analysis stems from a realisation that, for certain tasks, only a small portion of the results are needed at any one time and an analysis can make better use of resources if it only analyses the subset of the program that has bearing upon those results. This subset is discovered as the analysis progresses and the analysis incrementally requests solutions to goals, which may, in turn, request solutions to more goals [41, 42].

The approach is particularly suited to interactive applications such as development environments, where the user only needs analysis of the code with which they are interacting at any given time. For a more detailed description, see section 6.2.

2.5.6 $\top$

Our study of a monovariant flow analysis on a range of open source Python programs in chapter 7 shows that an average of 47% of the expressions were typed as $\top$, and that figure did not drop below 42% for any program. It may seem strange that an analysis that has all code available to it and which has no timeout on its execution would, essentially, abandon analysis almost half the time. The cause is a conspiracy of dynamic typing characteristics.

As we describe in detail in section 2.5.1, any tractable flow analysis approximates real execution and each approximation not only introduces imprecision but magnifies imprecision originating elsewhere. One particularly egregious example in dynamically typed languages are heterogeneous containers, which cause any value flowing into them to escape from the point of view of an object-insensitive analysis. Consider:

```
x = (1, "Hello", A())
```

Here x references a three-item immutable tuple, where each item has a different type: an integer, a string and a user-defined class instance. Without object sensitivity to distinguish this tuple instance from all others (see section 2.5.3), an item in a tuple must share a type with the other items at the same index in every other tuple.

But it gets worse, as many common Python collections are mutable:

```
x = [1, "Hello", A()]
x.sort()
```

Even object sensitivity cannot rescue flow analysis in this example. Where the three objects end up is decided by arbitrarily complex code in the three objects. This is undecidable, so the analysis must assume the values escape and any value pulled from a heterogeneous container is inferred as having type $\top$. The fact that heterogeneous containers are used heavily in Python programs goes some way towards explaining the high figure we quote above.

2.6 Bounding flow analyses

Flow analyses for higher-order languages do not scale well [34,37,46]. Monovariant propagation-based analyses, like Shiver's 0-CFA [36], that must calculate a dynamic transitive closure have cubic complexity in the size of the program. The cost of polyvariant analyses varies depending on how many extra abstract values they maintain. The k-CFA family are provably EXPTIME-complete for functional languages [24] and $O(n^{k+3})$ for object-oriented languages [29].

Cheaper and less-precise algorithms exist. Tip and Palsberg [46] question whether 0-CFA and above provide sufficient benefit to justify their cost. And yet the precision of 1-CFA—even more expensive at $O(n^4)$—was found to be "next to useless" [26] for a dynamically typed language. Spoon and Shivers [43] observe that analyses of statically typed languages, like that by Tip and Palsberg [46], can be precise without the expense of a polyvariant analysis because the static types from the type system bound the effect of any imprecision in the flow analysis, in particular, through the use of casts. In other words, when the flow analysis infers an imprecise concrete type, but the type system mandates a more precise interface type, the analysis can use the type system's judgement instead.

Figures 2.1 and 2.2 show our examples from sections 2.5.2 and 2.5.3 in the statically typed language Java rather than the dynamically typed language Python. Notice how the casts ensure the types of the source variables are no less precise than that of the destinations. Flow analyses for Java can use these to bound their result when the concrete type is inferred less precisely than the interface type. Jagannathan et al. [20] formalise such an approach in their flow analysis that uses the type system's static types to control the use of polyvariance. They prove that the analysis never assigns a type to a variable that is less precise than the type system's type assignment.

Of course, dynamically typed languages do not have a static type system. But that does not mean that they lack interface types they can take into account. Approaches, including our own, try to take advantage of these to produce the same effect.

```
class A {}
class B {}

class C {
    static Object identity(Object x) { return x; }
}

A a = new A();
A y = (A) C.identity(a);
B b = new B();
B z = (B) C.identity(b);
```

Figure 2.1: An example of parametric polymorphism in the statically typed language Java. Notice how the casts ensure the type of y is no less precise than that of a. Similarly for z and b. Flow analyses for Java can use these to bound their result when the concrete type fails to be inferred more precisely than the interface type.

In fact, we suggest that any flow analysis used for type inference in a higher-order language must bound these types occasionally by using other information to prevent an over-approximate abstract value causing the analysis to give up and return $\top$. This might happen when, for example, inferring the return type of a function whose abstract value includes non-callable values. Without dismissing those non-callable values out of hand, the analysis would be forced to return $\top$. We discuss this in more detail in section 3.2.

Adams et al. [1] present a flow analysis based on the $O(n)$-complexity sub-0-CFA to reduce the number of type checks needed in compiled Scheme. Their analysis regains some of the precision lost by the fast but inaccurate flow analysis by considering the *restrictive* operations performed on values. These operations act like type-checked casts and assert the type of the value after that point. Abstract values from the flow analysis that are incompatible with a restrictive operation can be dismissed from the type of a variable after it passes through the operation. We discuss this in detail in section 3.3. Tobin-Hochstadt and Felleisen [47] independently developed *occurrence typing*, which closely resembles the use of restrictive operations by Adams et al. [1], but they do not combine it with a flow analysis. Guha et al. [17] combine a flow analysis with an intra-procedural analysis of JavaScript type tags to produce a more precise result.

```
interface Transport {}
class Bus implements Transport {}
class Bicycle implements Transport {}

class Person {
    Person(Transport transport) {
        this.transport = transport;
    }
    Transport transport;
}

Bus redBus = new Bus();
Person jack = new Person(redBus);
Bus jacksBus = (Bus) jack.transport;

Bicycle shinyBike = new Bicycle();
Person jill = new Person(shinyBike);
Bicycle jillsBike = (Bicycle) jill.transport;
```

Figure 2.2: An example of data polymorphism in the statically typed language Java. Notice how the casts ensure the type of jacksBus is no less precise than that of redBus. Similarly for shinyBike and jillsBike. Generics would achieve something similar, but make the relationship between the person and the type of transport explicit, avoiding the need for casts. Flow analyses for Java can use these to bound their result when the concrete type fails to be inferred more precisely than the interface type.

Dynamically typed languages naturally permit a greater degree of polymorphism than their statically typed counterparts, making polyvariance particularly vital. But polyvariant analyses do not scale, so our work uses other sources of type information with a monovariant flow analysis to regain some of the advantages of polyvariance.

3 Approach

In this section we explain how two existing approaches exploit the way values are used in a program to infer concrete types that are more precise. The first uses the information simply to eliminate contradictions during flow analysis so that dependent expressions can be usefully typed. The second propagates the information to control-flow successors so that the types of later expressions can be refined in light of it.

We adapt this technique to a duck-typed language semantics and show that, by assuming a well-formed program, we can extend it to improve types at control-flow predecessors, not just at the control-flow successors.

3.1 Weaknesses of flow analysis

Type inference for dynamically typed languages cannot impose restrictions on the behaviour of the input programs, so it is usually based on a family of general program analyses [27, 36, 41, 42] that model data flow to determine what types of value can reach an expression [7]. These are known as concrete types [2,32]. Such flow analyses are also used with statically typed languages when a more precise type is needed than the interface types that the static type system provides [2, 32, 46, 48]. Being concrete-type analyses, they have the *potential* to infer a more precise type than the interface type, but there is no guarantee that they will. In fact, flow analyses work best with statically typed languages because the interface types prevent less-precise concrete type judgements propagating and polluting the analysis of the rest of the program [42].

Dynamically typed languages have no type system interface types to fall back on and, in their absence, flow analyses will propagate both precisely and imprecisely inferred concrete types from creation sites to the expressions of interest. The effect is that an imprecise type will quickly swamp the results as the imprecision explodes through the program and the analysis is forced to assume a conservative approximation.

The explosion is a result of the analysis being faced with contradictions as it is forced to reason about situations that never actually occur. We demonstrate this in section 1.4, where a flow analysis infers a slightly imprecise type for an expression, but that imprecision soon causes a contradiction because the method called on the expression does not exist in all the classes in the imprecise type. The pure flow analysis has no choice but to abandon type inference for the result of that method call and propagate $\top$ anywhere that that result might flow.

3.2 Solution: Impure flow analysis

Dynamically typed languages may not have interface types from a static type system with which to refine imprecise concrete types, but that does not mean they lack interface types altogether. The solution is to use other evidence of an expression's interface to recover an interface type which we use to bound the concrete type. Contradictions between the interface type and members of the concrete type can then be used to down-cast the concrete type for a more precise overall result.

This approach is already evident, to a limited extent, in the flow analysis for concrete-type inference in the duck-typed language Smalltalk by Spoon and Shivers [41, 42]. As usual, due to imprecision, their algorithm is sometimes forced to reason with contradictory information such as when inferring the return type of a method call on a variable whose concrete type includes a class that does not have such a method. Rather than giving up and inferring the largest type, $\top$, for the result of that call, they take advantage of the fact that the method call will only return when the object supports the method—the other situations report an error and do not return—by simply ignoring the classes in the concrete type that lack the method. As Spoon explains it, the technique guarantees *preservation* not *progress* and "type information is correct as long as the program continues executing but the program might nonetheless stop executing at any time" [41]. In particular, they note that even when an expression's type has been inferred as $\top$ they can still usefully type the result of methods called on the expression by using the method name to find all possible receivers [42].

What Spoon and Shivers have done is use evidence of the implicit interface at the expression to resolve the contradiction, in effect casting the expression's type to one that would guarantee progress, then proceeding with the analysis of the method return type on that basis. Their analysis is no longer purely based on data-flow approximation but now includes a crude approximation of a value's duck type.

However, they do not include the more-precise type in the inference result for the object's type nor in other results that depend on it. The cast is temporary and internal, just enough to usefully analyse the method-call.

3.3 Flow sensitivity from type tests

The technique is made explicit in the work of Adams et al. [1] in the context of the dynamically typed, functional language Scheme. Their approach is one of the few examples of a formalism explicitly developed to refine a flow analysis using other evidence of a value's type.

They look at the effect of *restrictive* operations that include an implicit type check on their parameter. The operations do not return if the type check fails so, if they do return, then the argument passed to them must have had the necessary type. This is similar to the unsupported-method scenario above except that, unlike Spoon and Shivers, Adams et al. propagate this "observational information" [1] in order to flow sensitively refine the type of the argument where it also appears at control-flow successors of the operation.

Taking Scheme pairs as an example, pairs are constructed using constructor **cons** or compound constructors such as **list** that can be implemented in terms of **cons** [40]. The pair `1 . 2` flows into variable x:

```
(let ((x (cons 1 2))) x)
```

As in other flow analyses for type inference, Adams et al. use constructors like this as a source of "constructive information" [1] that propagates through the program contributing to the concrete type at the expressions they reach.

Like any flow analysis, this is imprecise to some degree, especially as execution may depend on input, and the programmer is free to coordinate polymorphism along different paths using the same variable. In the following example the programmer relies on input being at an end so that the pair

1 . 2 flows to x rather than the integer 3.

```
(let ((x (if (eof-object? (read)) (cons 1 2) 3)))
  (cons (cdr x) (car x)))
```

Flow analysis, however, must be conservative and will propagate the constructive information from the pair constructor and integer constants to propagate both type pair and integer to the later instances of x passed to **cdr** and **car**.

It is at this point that Adams et al. do something novel: they use the "observational information" [1] provided by the call to **cdr**, a function that will not return if the argument passed to it is not a pair, to refine the type inferred for x at the call to **car**. The behaviours of this function mean that any value of x that reaches the call to **car** must be a pair.

Unlike Spoon and Shivers [42], who only use contradictions to recover a usable type for the result of the contradictory operation, Adams et al. propagate the observational information along with the constructive information resulting in a more-precise, flow sensitive type at control-flow successors.

Observational information comes from restrictive operations that happen before the expressions whose type they influence. But what happens before reaching the expression is only half the story.

3.4 Favouring well-formed programs

Adams et al. developed their analysis in order to optimise Scheme compilation by removing unnecessary run-time checks, for example, checking whether x is a pair for a second time in the call to **car** in the example above. This is a problem that, by definition, must allow for the possibility that programs might not be well formed and might halt with a type error at run time, which would be the case if the programmer had been wrong to assume input had been exhausted above. Development tools, on the other hand, typically insist that programs are well formed before they guarantee the correctness of their own assistance; consider, for example, a behaviour-preserving refactoring tool.

We take advantage of this extra freedom to improve the inferred types yet further in an effort to make types in well-formed programs as useful as possible for tools even if it means giving incomplete (or even incorrect) answers for ill-formed programs. Consider a modified example of the earlier example where, this time, we can see that the program is well-formed but the flow analysis cannot reach the same conclusion:

```
(let ((x (if (eof-object? (read)) (cons 1 2) 3)))
   (if (eof-object? (read)) (cons x (car x)) x))
```

Here the programmer is relying on domain-specific knowledge about the behaviour of **read**—that once it signals that it has reached the end of input, all subsequent calls to **read** will as well—to coordinate polymorphism such that only pairs reach the call to **car**. The flow analysis, which is conservative, cannot reason about this domain-specific knowledge and will use the constructive information to infer that x in the true branch of the second **if** condition might be an integer or a pair.

The call to **car** restricts the type of x to a pair after the call returns, but that is not helpful this time as, once the call returns, x is not used again. But what we can do is assume that the program is well formed, which means that x must be a pair both *at* **car** x and at all appearances of the same binding of x that precede that call.

3.5 Duck-typed language semantics

The approach of Adams et al. that we described in section 3.3 is for Scheme: a nominally typed rather than duck-typed language. We also sketch our extension to their approach in section 3.4 using Scheme to make the parallels clear. However, we believe it to be of greatest benefit with duck-typed languages and the remainder of this discussion, our formalism and our practical experience are all in that context.

The type information implied by the restrictive operations in Scheme is nominal: integer or not integer, pair or not pair. Each restrictive operation checks the type of its argument using its run-time type tag and each operation is monomorphic: only an exact type-tag match will suffice. Functions are pure, so no side effects are possible and, although Scheme has mutable variables, Adams et al. do not support observational information for those and rely on constructive information alone [1].

Our approach is complicated by the need to allow for imperative features, aliasing and mutable variables. We discuss the ideal duck-typed semantics in more detail in chapter 4, particularly in section 4.12, but for the sake of the discussion in this chapter, let us assume the following:

- Values are on the heap.
- Values have interfaces consisting of any number of abstract features.
- The set of features supported by a value is fixed when the value is created. Features cannot be added or removed later.
- Operations use values by requesting a feature that may or may not be present.
- Requesting a feature from a value performs an implicit *duck test*.
- Variables index and dereference a stack frame of heap locations.
- Variables can be updated to reference different values: they are mutable.

3.6 Contraindication

Recalling the discussion from section 3.1, our aim is to recover interface types that we can use to down cast the concrete types from flow analysis in the way that static types from the type system do for a statically typed language. We call this *contraindication*.

Concrete types are sets of abstract values, typically descriptions of the sources that create the run-time values such as constructors, constants, allocations or lambda abstractions. Because every value in the program belongs to exactly one of these sources, we refer to them as *classes*: disjoint categories of value. Until chapter 5 there is no assumption that these classes are object oriented, as is commonly the case. With this in mind, we define contraindication as follows:

Contraindication Down casting a concrete type using an interface type to eliminate from the former those classes that are contradicted by the latter.

We have seen two examples of existing work that make use of other information to refine the concrete type. The first, an analysis for a duck typed language, uses as little information as possible to refine the type just enough to allow the analysis to proceed. The second uses as much preceding information as possible to refine a type, but not in the context of a duck typed language. If we are to explore contraindication in a duck typed context, we must consider the information available in a duck typed language semantics.

Other than constructive information from class constructors, the main source of type information comes from *duck tests*. These are implicit tests that query a value for the presence of a feature before attempting to use it. The duck tests perform a similar role to the restrictive operations in Adams et al. [1] (section 3.3): values that pass the duck test must have the feature under test and those that fail must not. Unlike the constructive information, which produces nominal types, the duck tests expose aspects of a value's structural type. The key to contraindication in a duck typed language is finding a way to combine these two notions of type.

This is possible given a restriction on the definition of classes. They must define the *upper bound* of the features their values support. This allows us to use the presence of a feature implied by a duck test to *contraindicate* those classes that do not define that feature. Any contraindicated classes can be excluded from the concrete type because they cannot be the source of the value in any run of the program that passes the duck test.

3.6.1 Flavours of contraindication

Reasoning in terms of what happens when the program passes the duck test leads to different flavours of contraindication depending on which duck tests are used to contraindicate which types and what assumptions are made about the behaviour of the program.

- Like restrictive operations occurring before an expression (section 3.3), duck tests that precede the expression being typed lead to contraindication that is sound, even for ill-typed programs, as long as failing duck tests halt the program. This flavour is suitable for use in an optimising compiler, for instance.

- Alternatively, if the application makes it reasonable to assume a duck test never fails, duck tests that precede the expression being typed lead to contraindication that is sound regardless of whether duck tests halt the program. This flavour is suitable for applications such as an IDE or code documentation.

- Once we assume a duck test never fails, the duck tests on or after the expression being typed also lead to sound contraindication.

Contraindication produces the most precise concrete type from the most precise interface type—the largest set of duck tests. But we must be careful not to get carried away when deciding which duck tests to include: not every feature that appears to contribute to an expression's type actually does so. The types we want to infer are a *static* over-approximation of the values that may appear at an expression, so only those features that restrict the expression's type on every run of the program can be included in its type. We need to freeze the duck type to produce an interface type from only those duck tests that are guaranteed to occur.

In the next two sections we define precisely which features to include in the frozen duck type. In section 3.10 we outline a computable analysis to collect both kinds of feature based on intra-procedural data-flow analysis. In chapter 4 we formalise both the frozen duck types, the analyses to compute them and we prove their soundness properties.

3.7 Observed features

As discussed above, requesting a feature from a value triggers a duck test, and that test only succeeds if the value possesses the feature. When a value is always tested in this way *before* it reaches an expression, we know the value will have the feature at the expression if:

1. the language semantics is such that failing duck tests halt execution, in which case a value without the feature cannot reach the later expression; or

2. the input program is known never to fail a duck test, in which case the value is bound to have the feature before it arrives at the duck test.

This leads to the following definition of observed features:

Observed features The observed features of an expression are those whose absence from a value would cause the value to fail a duck test at a point in the program prior to the expression.

In the following fragment of Python, method f is an observed feature of x_{42}. No matter what happens outside this fragment, every value that reaches expression x_{42} will have a method f:

```
41  x.f()
42  print x
```

However, if the fragment is adapted as follows, method f is no longer an observed feature of x_{42} unless today is always Tuesday which, if the omitted code is sensible, is not going to be the case:

```
40  if today == Tuesday:
41      x.f()
42  print x
```

Readers with a static typing background might find it perverse to say that f is not required at line 42; surely the programmer must realise that Tuesday is a possibility and so must assign a value x with a method f, even if it is not always called?

Certainly the programmer must realise that Tuesday is a possibility and in a statically typed language that would, indeed, imply that they must assign a value x with a method f. However, this implication is imposed by the static type system to make the problem decidable; it is not required by logic. In a dynamically typed language, it is very possible the programmer coordinated polymorphism (section 1.3) in such a way that, whenever Tuesday arises, x has a method f but *not necessarily otherwise*:

```
31  class HasF:
32      def f(self):
33          print "Hello"
34
35  if today == Sunday:
36      x = 0
37  else:
38      x = HasF()
39
```

```
if today == Tuesday:
    x.f()
print x
```

This fragment is well formed, assuming Tuesday and Sunday are mutually exclusive; it will never encounter a failed duck test. However, observed features are sound even if the program is not well formed, assuming that failing a duck test halts execution.

It might appear that only statements whose execution *strictly dominates* an expression can be in the latter's observed features; after all, the other statements might not necessarily execute before reaching the expression, just as in the example. That is almost true and, indeed, it is how we calculate an approximation of observed features (section 3.10.2) but it is not quite accurate. Consider the following:

```
71 if today == Tuesday:
72     x.g()
73     x.f()
74 else:
75     x.f()
76     x.h()
77 print x
```

Neither line 73 nor line 75 dominate line 77 and yet method f is an observed feature of x. One way to explain it is that the *duck test* must dominate, even if the statements where the tests occur do not dominate individually. The situations can be much more subtle than the example above with dominating feature's tests being far removed from each other, even indirected through procedure calls. Our current implementation (chapter 6) does not attempt to consider such features although it remains an interesting avenue for future work (section 8.3.2).

Before moving on to consider tests occurring after the expression, it is important to note that these judgements rely on an aspect of our language semantics that we have been assuming until now (section 3.5):

> The set of features supported by a value is fixed when the value is created. Features cannot be added or removed later.

If this were not the case, a feature could be removed from a value between the duck test and the expression. Observed features would no longer be a sound approximation of the features at the expression.

3.8 Required features

While observed features may be sound even for ill-formed programs when failing duck tests halt a program, we assume a well-formed program in order to benefit from a second category of features, required features, which are sound only given that assumption. Our results in chapter 7 show that this second category accounts for the largest part of the improvement in type precision.

Required features The required features of an expression are those whose absence from a value at the expression will lead to a failing duck test on any run of the program including the expression.

This informal definition implies a causal relationship between the absence of the feature and the type error, but what it does not make clear is that the run-time type error need not happen immediately. The point at which the feature is requested may be far removed from the expression being typed, but the typing can still take advantage of it as long as the type error is inevitable.

In the following fragment of Python, method f is a required feature of both x_{77} and x_{78}. Any value in x that survives the duck test on line 78 will have had a method f in x on line 77. In other words, if the program is well formed, x_{78} has a method f.

```
77  print x
78  x.f()
```

As with observed features, if we adapt the fragment to make it input-dependent whether the feature is requested, method f is no longer a required feature of x_{77}:

```
77  print x
78  if today == Tuesday:
79      x.f()
```

Only those features that will *definitely* lead to a run-time type error if they are missing at an expression will be required features. Again, although this may seem perverse to readers with a static typing background, where types are used to tell the programmer if a missing feature *may* lead to an error, a dynamically typed language allows the programmer to coordinate polymorphism freely so required features have to be defined more strictly.

Our required features are similar to the success typings of Lindahl and Sagonas [23], which we describe in section 2.4. They define success typings as type signatures that statically over-approximate the set of arguments that will allow a function to return an argument, and over-approximates the set of values the function can return. In other words, all arguments not covered by the type signature will cause the function to encounter a run time type error and never return.

Lindahl and Sagonas [23] infer their typings from an analysis of pattern matches applied to values. These checks assert either exact value equality or that a value is a member of a particular named category of values. If the analysis can show that the assertion holds for any successful execution, the requirements of the match become part of the success typing. That means a value appearing at an expression but not included in the expression's success type must be guaranteed to fail a pattern match at some point during every run of the program.

In our case, any value appearing at an expression missing a feature listed in our approximated set of required features must be guaranteed to eventually be asked for one of those features on any run of the program, causing a duck test to fail.

3.9 Aliasing

Until now our examples have not included variable mutation, but any analysis we might hope to apply to a language like Python must. A duck test can only influence the features of an expression, be they observed or required, if the duck test requests the feature from the value at the expression.

In the languages targeted by Adams et al. [1] and Lindahl and Sagonas [23], Scheme and Erlang respectively, immutable variables combined with a value-based language semantics means that the issue does not arise: variables are defined once, so will always 'alias' themselves while nothing else aliases anything. In our more general language semantics (section 3.5) expressions reference values on the heap, which may be aliased, and a program can rebind a variable to point to a different value than the one with which it was initialised.

Exactly which expressions can be considered for observed or required features is subtle. Any of the expression's so called *may-aliases* [33] might yield the same value, but this is not strong enough. Consider the following:

```
19  x.f()
20  if today == Tuesday:
21      x = 10
22  print x
```

Here x_{19} may-aliases x_{22}, so method f is not an observed feature of x_{22}. Likewise:

```
7   print x
8   if today == Tuesday:
9       x = 10
10  x.f()
```

Here x_{10} may-aliases x_{7}, so method f is not a required feature of x_{7}.

The solution may appear to be *must-aliasing*, a much stronger guarantee:

"Names a and b are said to *must-alias* each other at a program point if, for all paths P from the program beginning to the program point, a and b both refer to the same location after execution along P" [33].

Indeed, we use an even stronger approximation of must-aliasing in our implementation because it is easy to calculate (section 6.5), but must-aliasing is stronger than necessary and excludes some observed and required features. For example, it prevents taking into account features that are requested at an expression that aliases our expression on every run of the program but not throughout the entire execution of the program. Imagine for a moment that Python had **do**/**while** loops:

```
34  y = ...
35  y.p()
36  do:
```

```
37      y.q()
38      y = ...
39  while x
```

In this example method q is a required feature of y_{34} and y_{35}. After line 38 y is no longer guaranteed to reference the same value, and its features may be different, but that does not change the fact that there is no path that can pass through y_{35} without encountering a duck test for method q on the same value.

Similarly, for observed features:

```
84  def fun(x):
85      x.f()
86
87  a = W()
88  fun(a)
89  a.g()
90
91  b = Z()
92  fun(b)
```

Here, method f is an observed feature of a_{88} because there is no path through this fragment that will reach line 88 if a does not have a method f; the duck test inside the function call would prevent it. However, must-aliasing would exclude x_{85} from consideration because it also gets bound to b in an unrelated situation.

All we require is that at run time the feature at the candidate expression is requested from the same value as the expression being typed. It does not mean that it cannot be requested from other values at the candidate expression at other times.

3.10 Computable analysis

Observed and required features are an uncomputable ideal. They are defined in terms of actual execution, so any useful analysis can, at best, approximate them. The issues discussed above concerning domination and aliasing are symptoms of this. The aim is to find a provably sound computable approximation without it being so conservative that it excludes scenarios likely to make a useful contribution in practice.

In chapter 4 we describe this in detail by repeatedly approximating observed and required features until we arrive at an analysis that is computable on an intra-procedural control flow graph using standard compiler algorithms for imperative languages. We sketch these analyses now.

3.10.1 Basic set

The principle behind both analyses is that languages make some duck tests *syntactically evident* from the source code. From these, we can derive a *basic set* of observed and required features and then propagate these through the program. In our Python-style ideal language the syntactically evident duck tests are method calls: `e.m()`

Syntactically evident A syntactic form that guarantees a duck test for a feature and unequivocally identifies both which feature is being requested and the syntactic element from whose value it is requested.

By this definition, expression `e` is the syntactic element whose value is requested for a feature and the feature requested is a method `m`.

The basic set of required features follows directly from the syntactically evident duck tests: a test for a feature means that feature is required at the expression being tested. The situation is slightly more complicated for observed features where a duck test implies an observed feature for the expressions that always follow and alias the tested expression, rather than the tested expression itself. We formalise this in section 4.7.2.

3.10.2 Propagation

The observed features of one node are also observed features of its *strictly dominated aliases*. Likewise, the required features of a node are also required features of its *postdominated aliases*. In this way, we can propagate the basic sets to their dominated and postdominated aliases. This is the basis of the analyses described in chapters 4 to 6, where we define dominating and postdominating aliasing and also how to approximate them, as they are uncomputable in general.

(Post)dominating aliases still exclude some situations we would rather include, for instance the example in section 3.7 where a feature is requested of the same value on both branches of a conditional: the feature (post)dominates other expressions but the syntactically evident statements do not. In the future we hope to improve this (section 8.3.2).

3.10.3 Using the approximated features

The observed and required features are useful on their own. We can use them to automatically generate API documentation, to suggest code completions and even to find some bugs. But our main aim, and the reason for calculating them in the first place, is to use them to improve the precision of flow-based type inference. We call this *contraindication* and, as we describe in section 3.6, is a matter of using the features of an expression to filter out any class that does not support the feature from the concrete type inferred by flow analysis. Now we know how to approximate those features and, as the approximation under-estimates the expression's features, using them for contraindication errs on the side of not contraindicating member of the concrete type. The result is that, given a sound concrete type as input, contraindication with our approximated features will yield a sound, and possibly more precise, concrete type.

In this section we have seen two examples of existing work that refine a concrete type from a flow analysis using non-constructive type information. We show how we can extend this by assuming a well-formed program to take more information into account. We map these concepts to a duck typed language semantics in the form of observed and required features and discuss the challenges presented by an imperative language. Finally, we sketch a computable approximation enabling sound contraindication of concrete types. In the next chapter we formalise the analyses laid out in this chapter and in chapter 5 we explore the challenges of applying these ideas to a concrete language, Python.

4 Formal presentation

In this chapter we formalise the ideas and theories discussed in the previous chapters.

The presentation is in two parts: firstly, the formalisation of observed and required features, leading to computable analyses to recover them (sections 4.5 to 4.10); secondly the contraindication analysis that refines concretes types using features (section 4.11).

4.1 Soundness properties

For the contraindication analysis, $\vdash_{\mathrm{CI}}$ (definition 26 in section 4.11), we establish the soundness property that it only produces valid concrete types, $\models_{\mathrm{C}}$ (definition 24 and theorem 10 in section 4.11):

$$\vdash_{\mathrm{CI}} \quad \Rightarrow \quad \models_{\mathrm{C}}$$

The contraindication analysis takes valid concrete types and 'had' features, $\models_{\mathrm{F}}$ (definition 10 in section 4.5.1), as input. Therefore, to enable our feature recovery analyses to be used for contraindication, we establish the soundness property that they only produce 'had' features.

For clarity, we establish soundness of the analyses in two steps. First, we prove that required features, $\vdash_{\mathrm{RF}}$ (definition 9 in section 4.5), and observed features, $\vdash_{\mathrm{OF}}$ (definition 11 in section 4.6), are 'had' features in well-formed programs (theorems 1 and 2):

$$\vdash_{\mathrm{RF}} \quad \Rightarrow \quad \models_{\mathrm{F}}$$

$$\vdash_{\mathrm{OF}} \quad \Rightarrow \quad \models_{\mathrm{F}}$$

Then we show that the analyses conservatively approximate observed and required features. This step is sketched in section 4.3.

Neither contraindication nor feature recovery are complete. There are valid concrete types that are not revealed through contraindication and 'had' features that are not recovered by our feature analyses.

4.2 Preliminaries

Let us dispense with various preliminaries that establish the framework for our formalism.

The relations in this chapter range over various sets that we define here. First we define S, the sets used by all parts of the formalism, followed by S^+, the additional sets needed to reason about contraindication. These sets are independent of a specific language semantics. In section 4.10 we extend these with S^π which is specific to the toy language presented in that section.

$$\boxed{\mathsf{S}}$$

$$\begin{array}{rcll}
P \in \textbf{Program} & & & \text{programs} \\
pc \in \textbf{Counter} & = & \mathbb{N} & \text{program counters} \\
m \in \textbf{Feature} & & & \text{features} \\
\nu \in \textbf{Value} & = & \textbf{Feature} \to \ldots & \text{values} \\
x, y \in \textbf{VarId} & & & \text{variable names} \\
\iota \in \textbf{Addr} & = & \{\iota_i | i \in \mathbb{N}\} & \text{locations} \\
\chi \in \textbf{Heap} & = & \textbf{Addr} \to \textbf{Value} & \text{heaps} \\
\varphi \in \textbf{Frame} & = & \textbf{VarId} \to \textbf{Addr} & \text{stack frames} \\
\sigma \in \textbf{Stack} & = & \textbf{Frame}^* & \text{stacks} \\
c \in \textbf{Configuration} & = & \textbf{Counter} \times \textbf{Stack} \times \textbf{Heap} & \text{configurations} \\
p \in \textbf{Path}_N & = & N^* & \text{paths}
\end{array}$$

$$\boxed{\mathsf{S}^+}$$

$$\begin{array}{rl}
n \in \textbf{Class} = \textbf{Feature} \to ... & \text{abstract values} \\
t \in \textbf{ConcreteType} = \mathcal{P}\left(\textbf{Class}\right) & \text{flow-based type}
\end{array}$$

For each of the sets in S we define the type of values they may contain, with the exception of **Feature**, **VarId**, **Program** and the range to which values map features. These depend on the language being modelled. For many of these sets we also include meta variables that are used throughout the chapter to range over members of the set.

We abuse the meta variable representing stacks, σ, so that, as well as being a sequence of stack frames, it is a function that take a variable and dereference it to an address, using the top stack frame.

$$\begin{array}{ll} \sigma(x) = \varphi(x) & \text{if } \sigma = \sigma'.\varphi \\ \quad\quad undefined & \text{otherwise} \end{array}$$

We use the function dom to extract the domain of a mapping in the conventional way. For example, where ν is a value, $\mathrm{dom}(\nu)$ returns the set of features in the value's domain.

Paths are constructed with the concatenation operator as in $p_1 \cdot p_2$, which results in a path made of all the nodes in p_1 followed by all the nodes in p_2.

4.2.1 Small step evaluation

We reason using a small step operational semantics. The specifics of any particular semantics is left until section 4.10. For the moment, let us just focus on the language-independent aspects of our approach.

The semantics of execution is defined as a transition from one configuration of a program to another in a single step. Configurations consists of a program counter, a stack and a heap allowing us to model imperative languages with side effects.

Definition 1 (Execution) $\rightsquigarrow$

$$P \vdash \langle pc, \sigma, \chi \rangle \rightsquigarrow \langle pc', \sigma', \chi' \rangle$$

The full definition of execution depends on the semantics of the language, an example of which we give in section 4.10. For now we just assume it may be input-dependent.

Frequently we need to reason about a configuration resulting from an arbitrary (possibly zero) number of steps. For this purpose we define $\rightsquigarrow^*$, the reflexive, transitive closure of execution.

Definition 2 (Transitive execution) $\rightsquigarrow^*$

$$P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow^* \langle pc', \sigma', \chi'\rangle$$

iff

$$\frac{pc = pc' \qquad \sigma = \sigma' \qquad \chi = \chi'}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow^* \langle pc', \sigma', \chi'\rangle}$$

$$\frac{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc', \sigma', \chi'\rangle}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow^* \langle pc', \sigma', \chi'\rangle}$$

$$\frac{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow^* \langle pc'', \sigma'', \chi''\rangle \qquad P \vdash \langle pc'', \sigma'', \chi''\rangle \rightsquigarrow^* \langle pc', \sigma', \chi'\rangle}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow^* \langle pc', \sigma', \chi'\rangle}$$

We assume that all executions of any program start from a single, identifiable configuration

Definition 3 (Initial configuration) init

$$\mathsf{init} = \langle 1, \sigma_1, \chi_1\rangle$$

and ends in a member of a set of identifiable final configurations, final.

Execution is input-dependent so $P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc', \sigma', \chi'\rangle$ means it is *possible* for configuration $\langle pc, \sigma, \chi\rangle$ to transition to configuration $\langle pc', \sigma', \chi'\rangle$ in program P but it is not guaranteed to. Therefore we define the paths of a program as all possible sequences of configurations from the initial configuration to one of the final configurations.

Definition 4 (Execution paths) Paths

$$\text{Paths} : \mathbf{Program} \to \mathcal{P}\left(\mathbf{Path_{Configuration}}\right)$$

$$\text{Paths}(P) = \left\{ p \;\middle|\; \begin{array}{c} p \in \text{ReachingPaths}(P) \\ \exists p', c.\, p = p' \cdot c \wedge c \in \mathsf{final} \end{array} \right\}$$

where

$$\text{ReachingPaths}(P) =$$
$$\mathsf{init} \cup \left\{ p \cdot \langle pc, \sigma, \chi \rangle \cdot \langle pc', \sigma', \chi' \rangle \;\middle|\; \begin{array}{c} p \cdot \langle pc, \sigma, \chi \rangle \in \text{ReachingPaths}(P) \\ P \vdash \langle pc, \sigma, \chi \rangle \rightsquigarrow \langle pc', \sigma', \chi' \rangle \end{array} \right\}$$

As short hand, we use $P \vdash \mathsf{init} \rightsquigarrow^* c$ to describe a configuration being reachable and $c \in p$ to mean that a configuration appears within a path. They are equivalent to the more verbose path form and we use them interchangeably without necessarily referencing the following lemmas.

Lemma 1.

$$\forall P, c\,(P \vdash \mathsf{init} \rightsquigarrow^* c \iff \exists p \in \text{Paths}(P), p_1, p_2.\, p = p_1 \cdot c \cdot p_2)$$

Lemma 2.

$$\forall p, c\,(c \in p \iff \exists p_1, p_2.\, p = p_1 \cdot c \cdot p_2)$$

4.2.2 (Post)domination

We are particularly interested in configurations that are *guaranteed* to precede or follow another; the *dominators* and *postdominators.*

A configuration dominates another only when it occurs on all paths through the program reaching the second configuration. A program point dominates another if it appears in some configuration on all paths reaching any configuration at the second program point.

A configuration postdominates another only when it occurs on all terminating paths through the program from the second configuration. A program point postdominates another if it appears in some configuration on all terminating paths through the program that include the second program point in some configuration.

We give a general definition of domination and postdomination over sets of paths as the relationship between sets of paths and their respective (post)dominators helps us to establish a computable approximation later (section 4.9.1).

Definition 5 (Domination over sets of paths) doms

$$\text{doms} \;:\; \mathbf{Path}_N \to \mathcal{P}\,(N \times N)$$

$$(n, n') \in \text{doms}(Q)$$

iff

$$\forall p \in Q, p_1, p_2\,(p = p_1 \cdot n' \cdot p_2 \;\Rightarrow\; \exists p_3, p_4.\; p_1 = p_3 \cdot n \cdot p_4)$$

Definition 6 (Postdomination over sets of paths) postdoms

$$\text{postdoms} \;:\; \mathbf{Path}_N \to \mathcal{P}\,(N \times N)$$

$$(n, n') \in \text{postdoms}(Q)$$

iff

$$\forall p \in Q, p_1, p_2\,(p = p_1 \cdot n' \cdot p_2 \;\Rightarrow\; \exists p_3, p_4.\; p_2 = p_3 \cdot n \cdot p_4)$$

4.3 Overview

We begin by giving an overview of the relations we define in this chapter. We describe the essence of each relation in English, and give diagrams that illustrate how the relations, individually or in combination, conservatively approximate others. Solid arrows in the diagrams indicate implications; solid arrows with multiple tails indicate that the conjunction of the relations at the tails imply the relation at the head. Ultimately, the relations combine to guarantee that contraindication produces valid concrete types.

$$\begin{tikzcd}
\boxed{P, pc \models_{\mathrm{C}} x : t} \\
P, pc \vdash_{\mathrm{CI}} x : t \arrow[u] \\
t \text{ from filtering } t' \text{ using } P, pc \models_{\mathrm{F}} x : m \arrow[u, dotted] \\
P, pc \models_{\mathrm{C}} x : t' \arrow[u, dotted, dash] \\
P, pc \vdash_{\mathrm{CFA}} x : t' \arrow[u]
\end{tikzcd}$$

Figure 4.1: Sketch of contraindication, $\vdash_{\mathrm{CI}}$. The analysis makes use of any available valid feature information, $\models_{\mathrm{F}}$, to refine concrete types from flow analysis, $\vdash_{\mathrm{CFA}}$. If the flow analysis types are valid $\models_{\mathrm{C}}$, the refined concrete types are valid as well, so the analysis is sound.

Valid concrete type $\boxed{P, pc \models_{\mathrm{C}} x : t}$ Concrete types are sets of abstract values that cover all the values appearing at a variable. Our overall goal is to infer valid concrete types that cover the values as precisely as possible, in other words making set t as small as possible.

We obtain a sound approximation of concrete types from flow analysis and then use contraindication to make these types more precise (figure 4.1).

Flow analysis $\boxed{P, pc \vdash_{\mathrm{CFA}} x : t}$ Flow-analysis-based type inference produces concrete types. We do not define a particular flow analysis, but it is assumed to exist and to be sound.

Contraindicated types $\boxed{P, pc \vdash_{\mathrm{CI}} x : t}$ Contraindication produces a more precise concrete type for a variable, given its concrete type from flow analysis and its interface type. The contraindicated type remains a valid concrete type so contraindication is sound.

A variable's interface type—its frozen duck type—consists of any number of *features*.

$$\begin{tikzcd} & & {P, pc \models_{\mathrm{F}} x : m} & & \\ {P, pc \vdash_{\mathrm{OF}} x : m} \arrow[r, dash] & {} \arrow[ur] \arrow[r, dash] & P\diamond \arrow[r, dash] & {} \arrow[ul] \arrow[r, dash] & {P, pc \vdash_{\mathrm{RF}} x : m} \end{tikzcd}$$

Figure 4.2: Sketch of soundness properties for observed and required features, $\vdash_{\mathrm{OF}}$ and $\vdash_{\mathrm{RF}}$. Both are sound for well-formed programs, $P\diamond$.

Having a feature $\boxed{P, pc \models_{\mathrm{F}} x : m}$ A variable is said to have a feature at a point in a program if its value always possesses the feature at that point.

Features are the subject of duck tests. The presence of duck tests implies the presence of features under different assumptions.

Duck test $\boxed{P, pc, \sigma \vdash_{\mathrm{DT}} \iota : m}$ Every attempt to use a feature of a value is preceded by a duck test that verifies that the value possesses the feature. If it does not, the duck test transitions the program to one of a set of identifiable error configurations.

We define two categories of feature that are implied by the duck tests in different ways (section 4.3).

Observed features $\boxed{P, pc \vdash_{\mathrm{OF}} x : m}$ Features that are based on duck tests occurring prior to a value reaching a variable. Assuming that duck tests halt execution when they fail, such tests imply the presence of features soundly regardless of the behaviour of the input program. Alternatively, assuming the program is known never to fail a duck test, the features are guaranteed present in well-formed programs.

Required features $\boxed{P, pc \vdash_{\mathrm{RF}} x : m}$ Features that are based on duck tests occurring after a value has reached a variable. These features are guaranteed sound in well-formed programs but not ill-formed programs.

Well formed program $\boxed{P\diamond}$ A well formed program never fails a duck test. Making this assumption makes observed and required features sound approximations of actual features.

$$\begin{tikzcd} P, pc' \vdash_{\mathrm{SDT}} x' : m \arrow[r] & P, pc', \sigma \vdash_{\mathrm{DT}} \sigma(x') : m \arrow[r] & \boxed{P, pc' \vdash_{\mathrm{RF}} x' : m} \end{tikzcd}$$

Figure 4.3: A basic set of required features, $\vdash_{\mathrm{RF}}$, is implied directly by the presence of syntactically evident duck tests, $\vdash_{\mathrm{SDT}}$.

To make contraindication practical, we develop analyses to approximate observed and required features for an imperative, duck-typed language. Precise observed and required features are uncomputable so we approximate them in stages until we achieve a computable approximation. We prove that these approximations are still sound.

The analyses rely on the language making some duck tests *syntactically evident*.

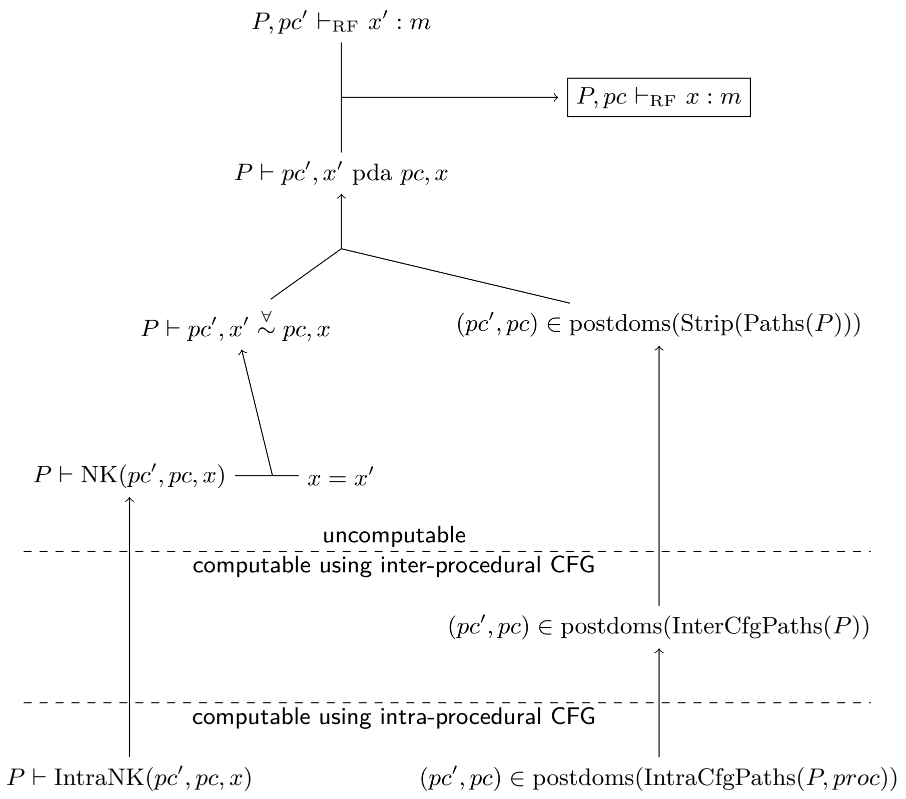
Syntactically evident duck tests $\boxed{P, pc \vdash_{\mathrm{SDT}} x : m}$ Some duck tests are dictated by the syntax. For example, in an object-oriented language the method call syntax fixes the feature being tested for and the variable whose value is tested.

We start with the approximation of required feature as the formalism is slightly simpler.

Required features A basic set of required features comes immediately from the syntax (figure 4.3). The remainder of our relations aim to propagate that basic set as far as possible (figure 4.4).

The first computable approximation we present depends on an interprocedural control flow analysis. This is precisely the kind of analysis whose inaccuracies in higher order languages we are attempting to bypass in the first place so we adapt the formalism and prove that it remains sound with only an intra-procedural flow analysis.

Postdominating aliases $\boxed{P \vdash pc', x' \text{ pda } pc, x}$ The glue that binds required features together. The features required by a variable at a program point are also required at other program points that it postdominates by the variables that it simultaneously aliases. This propagates the initial set of required features to points they postdominate but also propagates those new required features onwards.



$P, pc' \vdash_{\mathrm{RF}} x' : m$
$\boxed{P, pc \vdash_{\mathrm{RF}} x : m}$
$P \vdash pc', x'$ pda pc, x
$P \vdash pc', x' \overset{\forall}{\sim} pc, x$
$(pc', pc) \in \mathrm{postdoms}(\mathrm{Strip}(\mathrm{Paths}(P)))$
$P \vdash \mathrm{NK}(pc', pc, x)$
$x = x'$
uncomputable
computable using inter-procedural CFG
$(pc', pc) \in \mathrm{postdoms}(\mathrm{InterCfgPaths}(P))$
computable using intra-procedural CFG
$P \vdash \mathrm{IntraNK}(pc', pc, x)$
$(pc', pc) \in \mathrm{postdoms}(\mathrm{IntraCfgPaths}(P, proc))$

Figure 4.4: Propagating basic set of required features. At the top of the figure we see that members of the basic set of required features, derived directly from syntax (figure 4.3), propagate to other program points for which their variable is a postdominating alias, pda. Postdominating aliasing is uncomputable and is approximated by a combination of must-aliasing, $\overset{\forall}{\sim}$, and postdomination over execution paths, $\mathrm{postdoms}(\mathrm{Strip}(\mathrm{Paths}(P)))$. Still uncomputable, we approximate those two relations with computable kill analysis, IntraNK, and postdomination over control-flow-graph paths, either inter-procedural, $\mathrm{postdoms}(\mathrm{InterCfgPaths}(P))$, or intra-procedural, $\mathrm{postdoms}(\mathrm{IntraCfgPaths}(P, proc))$.

Postdominators $\boxed{(pc', pc) \in \text{postdoms}(\text{Strip}(\text{Paths}(P)))}$ Configuration-agnostic postdomination. Postdominating aliases are defined in terms of the actual configuration (program state) at the point of domination. It does not lend itself well to developing a computable approximation. We break it into two, half of which is postdomination between program points alone. Less accurate but has a straightforward computable approximation. The two separate analyses together approximate the original postdominating aliases.

Must-aliasing $\boxed{P \vdash pc', x' \overset{\forall}{\sim} pc, x}$ The second half of the approximation. As the connection between program state and postdomination has been lost by breaking them apart, we use must-aliasing to ensure the approximation is sound when combining with configuration-agnostic postdomination.

Kill analysis $\boxed{P \vdash \text{NK}(pc', pc, x)}$ An approximation of must-aliasing for variables of the same name. For a given variable name at one program point, the same name will be a must-alias at another program point if no intermediate step has changed the its value. This is a very restricted form of must-aliasing but it is easy to approximate computably as makes it equivalent to kill analysis.

Inter-procedural CFG postdomination $\boxed{(pc', pc) \in \text{postdoms}(\text{InterCfgPaths}(P))}$ Our first computable approximation of domination. The inter-procedural control flow graph over-approximates execution so under-approximates postdomination. Depending on an inter-procedural analysis is not ideal for a higher order language especially when the point of our analysis was to avoid the need for a precise (and slow) inter-procedural algorithm.

Intra-procedural CFG postdomination $\boxed{(pc', pc) \in \text{postdoms}(\text{IntraCfgPaths}(P, m))}$ Our second computable approximation of domination. The intra-procedural control flow graph no longer over-approximates actual execution but we prove that the domination relationship is still preserved for program points within the same procedure.

Intra-procedural CFG kill analysis $\boxed{P \vdash \text{IntraNK}(pc', pc, x)}$ Kill analysis for program points in the same procedure. In our toy language (sec-

$$\begin{tikzcd}
P, pc' \vdash_{\mathrm{SDT}} x' : m \arrow[r] & P, pc', \sigma \vdash_{\mathrm{DT}} \sigma(x') : m \arrow[d, dash] & \\
 & {} \arrow[r] \arrow[d, dash] & \boxed{P, pc \vdash_{\mathrm{OF}} x : m} \\
 & P \vdash pc', x' \text{ da } pc, x &
\end{tikzcd}$$

Figure 4.5: A basic set of observed features, $\vdash_{\mathrm{OF}}$, is derived from syntactically evident duck tests, $\vdash_{\mathrm{SDT}}$, occurring on a variable's dominating aliases, da, rather than directly on the variable as in figure 4.3.

tion 4.10) this safely approximates must-aliasing despite only considering kills in the same procedure as our language semantics ensures the stack is protected from modification during calls.

Observed features Approximating observed feature is slightly more complicated as the basic set does not follow directly from the syntactically evident duck tests but already needs to be linked to the tests by dominating aliasing (figure 4.5).

Dominating aliases $\boxed{P \vdash pc', x' \text{ da } pc, x}$ The mirror image of postdominating aliases. This time it is the glue that binds observed features together and also turns syntactically evident duck tests into the basic set of observed features.

Propagating the basic set follows the same pattern as for required features except in terms of domination rather than postdomination (figure 4.6).

4.4 Duck tests

Duck tests are the basis of all our analyses, so here we define exactly what we assume their behaviour to be. The semantics of a language must have duck tests with this behaviour if this formalism is to apply. In the following definition and the remainder of the thesis, TROUBLE is a set of program configurations that result from failed duck tests. These configurations may or may not be final.

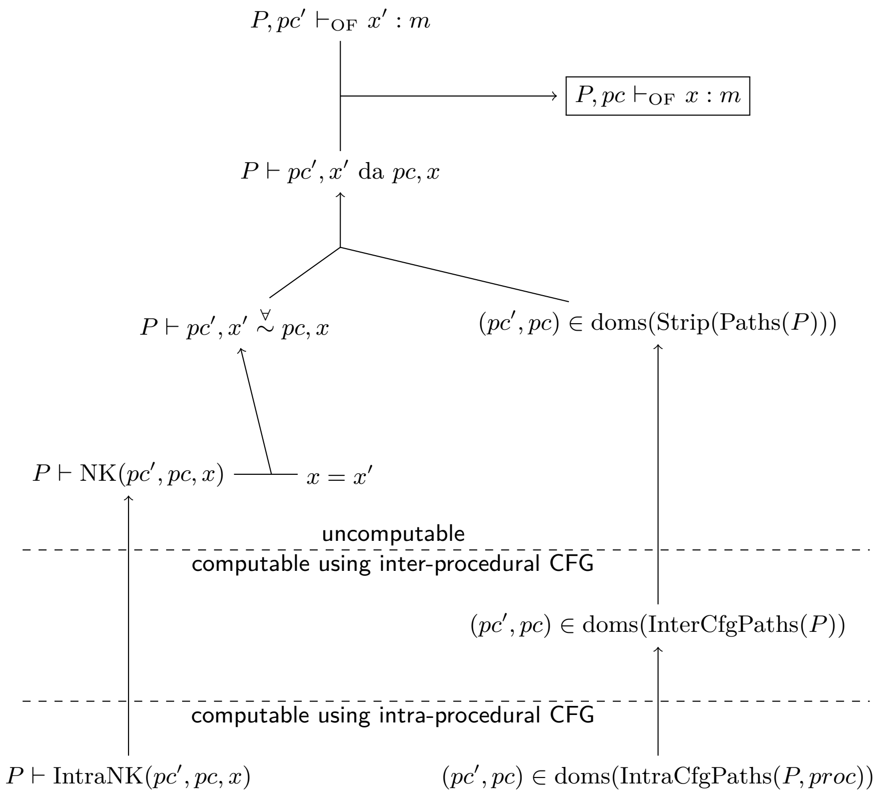

$P, pc' \vdash_{\mathrm{OF}} x' : m$
$\boxed{P, pc \vdash_{\mathrm{OF}} x : m}$
$P \vdash pc', x'$ da pc, x
$P \vdash pc', x' \overset{\forall}{\sim} pc, x$
$(pc', pc) \in \mathrm{doms}(\mathrm{Strip}(\mathrm{Paths}(P)))$
$P \vdash \mathrm{NK}(pc', pc, x)$
$x = x'$
uncomputable
computable using inter-procedural CFG
$(pc', pc) \in \mathrm{doms}(\mathrm{InterCfgPaths}(P))$
computable using intra-procedural CFG
$P \vdash \mathrm{IntraNK}(pc', pc, x)$
$(pc', pc) \in \mathrm{doms}(\mathrm{IntraCfgPaths}(P, proc))$

Figure 4.6: Propagating basic set of observed features. At the top of the figure we see that members of the basic set of observed features, derived as in figure 4.5, propagate to other program points for which their variable is a dominating alias, da. Dominating aliasing is uncomputable and is approximated by a combination of must-aliasing, $\overset{\forall}{\sim}$, and domination over execution paths, doms(Strip(Paths(P))). Still uncomputable, we approximate those two relations with computable kill analysis, IntraNK, and domination over control-flow-graph paths, either inter-procedural, doms(InterCfgPaths(P)), or intra-procedural, doms(IntraCfgPaths($P, proc$)).

Definition 7 (Duck test) $\vdash_{\mathrm{DT}}$

$$P, pc, \sigma \vdash_{\mathrm{DT}} \iota : m$$

iff

$$pc \notin \{pc_f \mid \langle pc_f, \sigma, \chi \rangle \in \mathsf{final}\}$$
$$\forall \chi\, (m \notin \mathrm{dom}(\chi(\iota)) \Rightarrow \forall c\, (P \vdash \langle pc, \sigma, \chi \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE}))$$

In sections 3.7 and 3.8 we said that observed and required features were sound for any well-formed program where well-formed programs are those that can never fail a duck test.

Definition 8 (Well formed program) $\diamond$

$$P\diamond$$

iff

$$\neg\exists c.\, P \vdash \mathsf{init} \rightsquigarrow^* c \wedge c \in \mathsf{TROUBLE}$$

Remember, this is not a statement of what a static analysis can establish about the program—we are not saying the program is well *typed*—just what the program will actually do. Given a sensible interpretation of the behaviour of the predicates, our example in section 1.3 is well formed by this definition.

4.5 Required features

In section 3.8 we defined required features informally. Now we do so formally. This definition is not concerned with how to calculate required features statically and, as such, is defined in terms of actual execution rather than any static approximation of real executions such as control flow graph succession.

Definition 9 (Required feature) $\vdash_{\mathrm{RF}}$

$$P, pc \vdash_{\mathrm{RF}} x : m$$

iff

$$\forall p \in \mathrm{Paths}(P), \sigma, \chi, p_1, p_2 \\ \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \right)$$

This definition says that a variable requires a feature at a point in a program if and only if, whenever that program point is reached on any run of the program, the value referenced by the variable will encounter a duck test for that feature, if not immediately then eventually.

Example. If P is the example from section 1.3 we could say

$$P, 17 \vdash_{\mathrm{RF}} \mathsf{p} : \mathsf{foo}$$
$$P, 17 \vdash_{\mathrm{RF}} \mathsf{p} : \mathsf{bar}$$

Example. Given a sensible interpretation of the predicates, we could also say

$$P, 14 \vdash_{\mathrm{RF}} \mathsf{p} : \mathsf{foo}$$

because definition 9 is about what actually happens rather than what a static analysis can prove happens.

Note that the feature need not always be required at the same variable on each execution; just that, on every execution, it must be required of some variable. For example, in the following program P it holds that $P, 1 \vdash_{\mathrm{RF}} \mathsf{x} : \mathsf{n}$.

```
1  x = ...
2  if x:
3      x.m()
4      x.n()
```

```
    x.q()
else:
    x.n()
```

4.5.1 Soundness

We judge that required features are sound if any variable—a static syntactic element—is guaranteed to have all its required features at run time. This is a static requirement, in other words it must hold for any run of the program.

Definition 10 (Having a feature) $\models_{\mathrm{F}}$

$$P, pc \models_{\mathrm{F}} x : m$$

iff

$$\forall \sigma, \chi\, (P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi\rangle \;\Rightarrow\; m \in \mathrm{dom}(\chi(\sigma(x))))$$

A variable can be said to have a feature, statically, at a program point if, whenever that program point is reached on any run of the program, the value referenced by the variable satisfies the requirements of the feature.

Now we prove that a required featured of a variable in a well formed program guarantees the variable always has the feature. The proof relies on an assumption about the semantics of the language that means that a value's features can not change at run time.

Assumption 1 (The set of features of a value is fixed within a program)**.**

$$\begin{gathered}\forall p \in \mathrm{Paths}(P), pc, x, \sigma, \chi, pc', x', \sigma', \chi', p_1, p_2, p_3\\ \left(\begin{array}{l} p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_3 \Rightarrow \\ \qquad\qquad \forall \iota \in \mathrm{dom}(\chi) \cap \mathrm{dom}(\chi').\, \mathrm{dom}(\chi(\iota)) = \mathrm{dom}(\chi'(\iota)) \end{array}\right)\end{gathered}$$

It is left to the particular operational semantics of the language to enforce this but in chapter 5 we will see a real world language whose semantics fail to do so but whose programming conventions are such that the results remain reasonable (chapter 7).

Theorem 1 (Required features are sound for well formed programs)**.**

$$\forall P, pc, x, m \left(\begin{array}{c} P\diamond \\ P, pc \vdash_{\mathrm{RF}} x : m \end{array} \Rightarrow P, pc \models_{\mathrm{F}} x : m \right)$$

Proof.

Given

$$P\diamond \tag{4.1}$$

$$P, pc \vdash_{\mathrm{RF}} x : m \tag{4.2}$$

Show

$$P, pc \models_{\mathrm{F}} x : m$$

By (4.1) and definition 8

$$\neg \exists c.\, P \vdash \mathsf{init} \rightsquigarrow^{*} c \wedge c \in \mathsf{TROUBLE} \tag{4.3}$$

By (4.2) and definition 9

$$\begin{array}{l} \forall p \in \mathrm{Paths}(P), \sigma, \chi, p_1, p_2 \\ \quad \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \right) \end{array} \tag{4.4}$$

Show, from definition 10

$$\forall \sigma, \chi \left(P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi \rangle \;\Rightarrow\; m \in \mathrm{dom}(\chi(\sigma(x))) \right)$$

Take σ, χ arbitrary.

Assume

$$\exists p \in \mathrm{Paths}(P), p_1, p_2.\, p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.5}$$

$$m \notin \mathrm{dom}(\chi(\sigma(x))) \tag{4.6}$$

Show contradiction.

Take p, p_1, p_2 such that by (4.5)

$$p \in \mathrm{Paths}(P) \tag{4.7}$$

$$p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.8}$$

By (4.8), (4.7) and (4.4)

$$\exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \tag{4.9}$$

Assume pc', σ', χ' such that by (4.9)

$$\langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.10}$$

$$P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \tag{4.11}$$

By (4.11) and definition 7

$$pc' \notin \{pc_f \mid \langle pc_f, \sigma, \chi \rangle \in \mathsf{final}\} \tag{4.12}$$

$$\forall \chi \, (m \notin \mathrm{dom}(\chi(\sigma(x))) \Rightarrow \forall c \, (P \vdash \langle pc', \sigma', \chi \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE})) \tag{4.13}$$

By (4.6), (4.10) and assumption 1

$$m \notin \mathrm{dom}(\chi'(\sigma(x))) \tag{4.14}$$

By (4.14) and (4.13)

$$\forall c \, (P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE}) \tag{4.15}$$

By (4.12), (4.10), (4.8), (4.7) and definition 4

$$\exists c.\, P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \tag{4.16}$$

By (4.15) and (4.16)

$$\exists c.\, P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \wedge c \in \mathsf{TROUBLE} \tag{4.17}$$

By (4.7)–(4.17) and (4.3)

$$\bot$$

$\square$

4.6 Observed features

Similar to required features, observed features are defined in terms of duck tests but this time the duck tests precede the program point in question. One again, the definition is not concerned with how to calculate the features statically and is defined in terms of actual execution.

Definition 11 (Observed feature) $\vdash_{\text{OF}}$

$$P, pc \vdash_{\text{OF}} x : m$$

iff

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2$$
$$\left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in p_1 \\ P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m \end{array} \right)$$

This definitions says that a variable observes a feature at a point in a program if and only if, whenever that program point is reached on any run of the program, the value referenced by the variable will already have encountered a duck test for that feature. Unlike with required features, duck tests *at* the program point are not included.

Example. If P is the example from section 1.3 we could say

$$P, 18 \vdash_{\text{OF}} \mathsf{p} : \mathsf{foo}$$

but

$$P, 18 \nvdash_{\text{OF}} \mathsf{p} : \mathsf{bar}$$

As with required features, the feature need not always be observed at the same variable on each execution; just that, on every execution, it must be observed of some variable.

4.6.1 Soundness

We judge that observed features are sound if any variable is guaranteed to have all its observed features at run time.

First we prove that observed features are sound in a well formed program.

Theorem 2 (Observed features are sound for well formed programs)**.**

$$\forall P, pc, x, m \left(\begin{array}{c} P\diamond \\ P, pc \vdash_{\mathrm{OF}} x : m \end{array} \Rightarrow P, pc \models_{\mathrm{F}} x : m \right)$$

Proof.

Given

$$P\diamond \tag{4.18}$$

$$P, pc \vdash_{\mathrm{OF}} x : m \tag{4.19}$$

Show

$$P, pc \models_{\mathrm{F}} x : m$$

By (4.18) and definition 8

$$\neg\exists c.\, P \vdash \mathsf{init} \leadsto^* c \wedge c \in \mathsf{TROUBLE} \tag{4.20}$$

By (4.19) and definition 11

$$\begin{array}{l} \forall p \in \mathrm{Paths}(P), \sigma, \chi, p_1, p_2 \\ \quad \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{r} \langle pc', \sigma', \chi' \rangle \in p_1 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \right) \end{array} \tag{4.21}$$

Show, from definition 10

$$\forall \sigma, \chi\, (P \vdash \mathsf{init} \leadsto^* \langle pc, \sigma, \chi \rangle \;\Rightarrow\; m \in \mathrm{dom}(\chi(\sigma(x))))$$

Take σ, χ arbitrary.

Assume

$$\exists p \in \mathrm{Paths}(P), p_1, p_2.\, p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.22}$$

$$m \notin \mathrm{dom}(\chi(\sigma(x))) \tag{4.23}$$

Show contradiction.

Take p, p_1, p_2 such that by (4.22)

$$p \in \mathrm{Paths}(P) \tag{4.24}$$

$$p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.25}$$

By (4.25), (4.24) and (4.21)

$$\exists pc', \sigma', \chi' . \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in p_1 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \tag{4.26}$$

Assume pc', σ', χ' such that by (4.26)

$$\langle pc', \sigma', \chi' \rangle \in p_1 \tag{4.27}$$

$$P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \tag{4.28}$$

By (4.28) and definition 7

$$\forall \chi \, (m \notin \mathrm{dom}(\chi(\sigma(x))) \Rightarrow \forall c \, (P \vdash \langle pc', \sigma', \chi \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE})) \tag{4.29}$$

By (4.23), (4.27), (4.25) and assumption 1

$$m \notin \mathrm{dom}(\chi'(\sigma(x))) \tag{4.30}$$

By (4.30) and (4.29)

$$\forall c \, (P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE}) \tag{4.31}$$

By (4.31), (4.27) and (4.25)

$$\exists c. \, P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \wedge c \in \mathsf{TROUBLE} \tag{4.32}$$

By (4.24)–(4.32) and (4.20)

$$\perp$$

□

We can also prove that observed features are sound if duck tests halt execution. This allows them to be used to reason about ill formed programs in languages with this behaviour making them suitable for applications such as optimisation.

Theorem 3 (Observed features are sound when failing duck tests are final and the program has not already failed).

$$\forall P, pc, x, m \left(\begin{array}{c} P, pc \vdash_{\mathrm{OF}} x : m \\ \forall c\,(c \in \mathsf{TROUBLE} \Rightarrow \neg\exists c'.\, P \vdash c \rightsquigarrow c') \\ \forall \sigma, \chi\,(P \vdash \mathsf{init} \rightsquigarrow^{*} pc, \sigma, \chi \Rightarrow \langle pc, \sigma, \chi\rangle \notin \mathsf{TROUBLE}) \end{array} \quad \Rightarrow \; P, pc \models_{\mathrm{F}} x : m \right)$$

Proof.

Given

$$P, pc \vdash_{\mathrm{OF}} x : m \tag{4.33}$$

$$\forall c\,(c \in \mathsf{TROUBLE} \Rightarrow \neg\exists c'.\, P \vdash c \rightsquigarrow c') \tag{4.34}$$

$$\forall \sigma, \chi\,(P \vdash \mathsf{init} \rightsquigarrow^{*} pc, \sigma, \chi \Rightarrow \langle pc, \sigma, \chi\rangle \notin \mathsf{TROUBLE}) \tag{4.35}$$

Show

$$P, pc \models_{\mathrm{F}} x : m$$

By (4.33) and definition 11

$$\begin{array}{l} \forall p \in \mathrm{Paths}(P), \sigma, \chi, p_1, p_2 \\ \qquad \left(p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'.\; \begin{array}{c} \langle pc', \sigma', \chi'\rangle \in p_1 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \right) \end{array} \tag{4.36}$$

Show, from definition 10

$$\forall \sigma, \chi\,(P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi\rangle \;\Rightarrow\; m \in \mathrm{dom}(\chi(\sigma(x))))$$

Take σ, χ arbitrary.

Assume

$$\exists p \in \mathrm{Paths}(P), p_1, p_2.\, p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \tag{4.37}$$

$$m \notin \mathrm{dom}(\chi(\sigma(x))) \tag{4.38}$$

Show contradiction.

Take p, p_1, p_2 such that by (4.37)

$$p \in \mathrm{Paths}(P) \tag{4.39}$$

$$p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \tag{4.40}$$

By (4.40), (4.39) and (4.36)

$$\exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in p_1 \\ P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \tag{4.41}$$

Assume pc', σ', χ' such that by (4.41)

$$\langle pc', \sigma', \chi' \rangle \in p_1 \tag{4.42}$$

$$P, pc', \sigma' \vdash_{\mathrm{DT}} \sigma(x) : m \tag{4.43}$$

By (4.43) and definition 7

$$\forall \chi \, (m \notin \mathrm{dom}(\chi(\sigma(x))) \Rightarrow \forall c \, (P \vdash \langle pc', \sigma', \chi \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE})) \tag{4.44}$$

By (4.38), (4.42), (4.40) and assumption 1

$$m \notin \mathrm{dom}(\chi'(\sigma(x))) \tag{4.45}$$

By (4.45) and (4.44)

$$\forall c \, (P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE}) \tag{4.46}$$

By (4.42) and (4.40)

$$\begin{array}{c} \exists p_3, p_4, c.\, p = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot c \cdot p_4 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \\ \vee \\ \exists p_3.\, p = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \end{array} \tag{4.47}$$

By (4.46) and (4.47)

$$\begin{array}{c} \exists p_3, p_4, c.\, p = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot c \cdot p_4 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \wedge c \in \mathsf{TROUBLE} \\ \vee \\ \langle pc, \sigma, \chi \rangle \in \mathsf{TROUBLE} \end{array} \tag{4.48}$$

By (4.39)–(4.48), (4.34) and (4.35)

$$\bot$$

□

Although sound, observed and required features are universally quantified over actual program execution making them uncomputable so now we develop a series of approximations that result in a computable static approximation.

The majority of the discussion in this chapter is independent of a particular language semantics—as long as the language has certain properties the techniques are applicable—but to complete the discussion we develop a dynamically typed toy language semantics with ideal properties and use it to demonstrate that a computable approximation is achievable in practice.

4.7 Static approximation

Our approach to approximating observed and required features is to calculate a *basic set* of features from *syntactically evident* duck tests (section 3.10.1). These guarantee that a variable's value will be tested for a feature on any execution that reaches the variable.

Definition 12 (Syntactically evident duck test) $\vdash_{\mathrm{SDT}}$

$$P, pc \vdash_{\mathrm{SDT}} x : m$$

iff

$$\forall \sigma, \chi \, (P \vdash \mathsf{init} \rightsquigarrow^* \langle pc, \sigma, \chi \rangle \Rightarrow P, pc, \sigma \vdash_{\mathrm{DT}} \sigma(x) : m)$$

4.7.1 Required features

The basic set of required features follows directly from these tests. For example, in an object oriented language like Python where a feature is a method, a method call makes the presence of that method immediately required at the call site. For example, $P, 43 \vdash_{\mathrm{SDT}} \mathsf{x} : \mathsf{m}$ and therefore $P, 43 \vdash_{\mathrm{RF}} \mathsf{x} : \mathsf{m}$ in the following P:

```
42  ...
```

```
x.m()
...
```

Lemma 3.

$$P, pc \vdash_{\text{SDT}} x : m \Rightarrow P, pc \vdash_{\text{RF}} x : m$$

The proof follows from definitions 9 and 12.

Under certain conditions a required feature can be propagated from one program point to another. This is the case when, by a conspiracy of post-domination, aliasing and language properties, guaranteed failure as a result of a feature's absence at one point guarantees eventual failure from the same absence at another point. This occurs when the execution of one point mandates the execution of a later one—postdomination—and a variable at one point refers to the same value as a variable at the other—aliasing—and is better expressed in the following definition.

Definition 13 (Postdominating alias) pda

$$P \vdash pc', x' \text{ pda } pc, x$$

iff:

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2 \\ \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \;\Rightarrow\; \exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \right)$$

Now that we have described the language requirements, let us state the theorem formally and prove it.

Theorem 4 (The features required by a variable at a program point are also required by the points for which it is a postdominating alias)**.**

$$\forall P, pc, x, pc', x', m \left(\begin{array}{c} P, pc' \vdash_{\text{RF}} x' : m \\ P \vdash pc', x' \text{ pda } pc, x \end{array} \Rightarrow P, pc \vdash_{\text{RF}} x : m \right)$$

Proof.

Given

$$P, pc' \vdash_{\text{RF}} x' : m \tag{4.49}$$

$$P \vdash pc', x' \text{ pda } pc, x \tag{4.50}$$

Show

$$P, pc \vdash_{\text{RF}} x : m$$

By (4.49) and definition 9

$$\begin{array}{l} \forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2 \\ \qquad \left(p = p_1 \cdot \langle pc', \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc'', \sigma', \chi'. \begin{array}{c} \langle pc'', \sigma', \chi' \rangle \in \langle pc', \sigma, \chi \rangle \cdot p_2 \\ P, pc'', \sigma' \vdash_{\text{DT}} \sigma(x') : m \end{array} \right) \end{array} \tag{4.51}$$

By (4.50) and definition 13

$$\begin{array}{l} \forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2 \\ \qquad \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \ \Rightarrow \ \exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \right) \end{array} \tag{4.52}$$

Show, from definition 9

$$\begin{array}{l} \forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2 \\ \qquad \left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m \end{array} \right) \end{array}$$

Take p, σ, χ, p_1, p_2 arbitrary.

Assume

$$p \in \text{Paths}(P) \tag{4.53}$$

$$p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \tag{4.54}$$

$$\neg \exists pc', \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m \end{array} \tag{4.55}$$

Show contradiction.

By (4.53), (4.54) and (4.52)

$$\exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \tag{4.56}$$

Assume σ', χ', p_3, p_4 such that by (4.56)

$$p_2 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \tag{4.57}$$

$$\sigma(x) = \sigma'(x') \tag{4.58}$$

By (4.57) and (4.54)

$$p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \tag{4.59}$$

By (4.59), (4.53) and (4.51)

$$\exists pc'', \sigma'', \chi''. \begin{array}{c} \langle pc'', \sigma'', \chi'' \rangle \in \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ P, pc'', \sigma'' \vdash_{\mathrm{DT}} \sigma'(x') : m \end{array} \tag{4.60}$$

By (4.60) and (4.57)

$$\exists pc'', \sigma'', \chi''. \begin{array}{c} \langle pc'', \sigma'', \chi'' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc'', \sigma'' \vdash_{\mathrm{DT}} \sigma'(x') : m \end{array} \tag{4.61}$$

By (4.61), (4.59) and assumption 1

$$\exists pc'', \sigma'', \chi''. \begin{array}{c} \langle pc'', \sigma'', \chi'' \rangle \in \langle pc, \sigma, \chi \rangle \cdot p_2 \\ P, pc'', \sigma'' \vdash_{\mathrm{DT}} \sigma(x) : m \end{array} \tag{4.62}$$

By (4.56)–(4.61) and (4.55)

$$\bot$$

□

Definition 13 says that a variable at a program point has a postdominating alias at a second program point if, whenever the first point is reached on any run of the program, the second point will be reached subsequently in a configuration where the value of the first variable at the first point is the same as the second variable at the second point. This does not preclude the possibility that the second point might *additionally* be reached in a configuration where the variables do not have the same value. In this way, postdominating aliases are weaker than must-aliases in which the configuration at the program point would be universally quantified:

$$\begin{aligned}&\forall p \in \mathrm{Paths}(P), \sigma, \chi, p_1, p_2\\ &\qquad\left(p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \;\Rightarrow\; \forall \sigma', \chi'. \exists p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array}\right)\end{aligned}$$

Consider the situation

```
1 y = ...
2 y.p()
3 do:
4     y.q()
5     y = ...
6 while x
```

In this example $P \vdash 4, \mathsf{y}$ pda $2, \mathsf{y}$ with definition 13 but not with the must-aliasing variant above.

However in other situations, the approximation misses necessary features. For example, in the following program P, it is easy to determine that $P, 3 \vdash_{\mathrm{RF}} \mathsf{x} : \mathsf{n}$ and $P, 5 \vdash_{\mathrm{RF}} \mathsf{x} : \mathsf{n}$. But $\neg P \vdash 3, \mathsf{x}$ pda $1, \mathsf{x}$ and $\neg P \vdash 5, \mathsf{x}$ pda $1, \mathsf{x}$ as neither line 3 nor line 5 postdominate line 1 even though x_3 and x_5 do alias x_1 so the fact that $P, 1 \vdash_{\mathrm{RF}} \mathsf{x} : \mathsf{n}$ is missed.

```
1 x = ...
2 if random():
3     x.n()
4 else:
5     x.n()
```

Despite this, the postdominating alias definition leads us towards a simple algorithm for a computable approximation so we leave refining it for future work section 8.3.2.

4.7.2 Observed features

Unlike the required features, the basic set of observed features does not follow directly from the syntactically evident duck tests but, instead, requires that the test be on a dominating alias the of the variable in the basic set. For example, $P, 43 \vdash_{\text{SDT}} \mathsf{x} : \mathsf{m}$ and $P, 44 \vdash_{\text{OF}} \mathsf{x} : \mathsf{m}$ but $P, 43 \nvdash_{\text{OF}} \mathsf{x} : \mathsf{m}$ and $P, 44 \nvdash_{\text{OF}} \mathsf{x} : \mathsf{n}$ in the following P:

```
...
x.m()
x.n()
...
```

Dominating aliasing is just the mirror image of postdominating aliasing:

Definition 14 (Dominating alias) da

$$P \vdash pc', x' \text{ da } pc, x$$

iff:

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2$$
$$\left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \;\Rightarrow\; \exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_1 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \right)$$

Lemma 4.

$$\begin{array}{c} P, pc' \vdash_{\text{SDT}} x' : m \\ P \vdash pc', x' \text{ da } pc, x \end{array} \Rightarrow P, pc \vdash_{\text{OF}} x : m$$

Proof.

Given

$$\forall \sigma, \chi \left(P \vdash \mathsf{init} \rightsquigarrow^* \langle pc', \sigma, \chi \rangle \Rightarrow P, pc', \sigma \vdash_{\text{DT}} \sigma(x') : m \right) \tag{4.63}$$

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2$$
$$\left(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \;\Rightarrow\; \exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_1 = p_3 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \right) \tag{4.64}$$

Show, from definition 11

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2$$
$$\left(p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \Rightarrow \exists pc', \sigma', \chi'. \begin{array}{c}\langle pc', \sigma', \chi'\rangle \in p_1 \\ P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m\end{array}\right)$$

Take p, σ, χ, p_1, p_2 arbitrary.

Assume

$$p \in \text{Paths}(P) \tag{4.65}$$
$$p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \tag{4.66}$$
$$\neg\exists pc', \sigma', \chi'. \begin{array}{c}\langle pc', \sigma', \chi'\rangle \in p_1 \\ P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m\end{array} \tag{4.67}$$

Show contradiction.

By (4.65), (4.66) and (4.64)

$$\exists \sigma', \chi', p_3, p_4. \begin{array}{c}p_1 = p_3 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_4 \\ \sigma(x) = \sigma'(x')\end{array} \tag{4.68}$$

Assume σ', χ', p_3, p_4 such that by (4.68)

$$p_1 = p_3 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_4 \tag{4.69}$$
$$\sigma(x) = \sigma'(x') \tag{4.70}$$

By (4.69) and (4.63)

$$P, pc', \sigma' \vdash_{\text{DT}} \sigma'(x') : m \tag{4.71}$$

By (4.71) and (4.70)

$$P, pc', \sigma' \vdash_{\text{DT}} \sigma(x) : m \tag{4.72}$$

By (4.68)–(4.72) and (4.67)

$$\perp$$

$\square$

Propagating observed features is the same as propagating required features except that domination replaces postdomination. We omit the proofs for brevity.

Theorem 5 (The observed features of a variable at a program point are also observed features of the points for which it is a dominating alias)**.**

$$\forall P, pc, x, pc', x', m \left(\begin{matrix} P, pc' \vdash_{\mathrm{OF}} x' : m \\ P \vdash pc', x' \text{ da } pc, x \end{matrix} \Rightarrow P, pc \vdash_{\mathrm{OF}} x : m \right)$$

4.7.3 Domination, postdomination and aliasing

We separate (post)dominating aliasing into (post)domination between program points and aliasing as, individually, they have existing standard compiler analyses. However, this loses some precision as aliasing no longer has access to the full configurations in the domination relation so we cannot be as precise about when the variable at the program counters has to alias. We must blindly insist that aliasing *always* be guaranteed, not just when it coincides with the domination.

Real execution paths include the complete configuration at each point so we also define an auxiliary function that converts the paths to simple sequences of program counters which we use when we are only interested to the relationship between executions and points in the source code.

Definition 15 (Configuration stripping) Strip

$$\text{Strip} : \mathcal{P}\left(\mathbf{Path_{Configuration}}\right) \to \mathcal{P}\left(\mathbf{Path_{Counter}}\right)$$

$$\text{Strip}(ps) = \{\text{StripOne}(p) \mid p \in ps\}$$

where

$$\text{StripOne}(\langle pc, \sigma, \chi\rangle) = pc$$
$$\text{StripOne}(\langle pc, \sigma, \chi\rangle \cdot p) = pc \cdot \text{StripOne}(p)$$

One program point postdominates another if, when the second is reached, the first will definitely be reached from it.

Lemma 5.

$$\forall P, pc, pc' \left(\begin{array}{c} (pc', pc) \in \text{postdoms}(\text{Strip}(\text{Paths}(P))) \\ \Leftrightarrow \\ \forall p \in \text{Paths}(P), p_1, p_2, \sigma, \chi \left(\begin{array}{l} p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \Rightarrow \\ \\ \qquad \exists \sigma', \chi'.\ \langle pc', \sigma', \chi' \rangle \in p_2 \end{array} \right) \end{array} \right)$$

Definition 16 (Must-aliasing) $\overset{\forall}{\sim}$

$$P \vdash pc, x \overset{\forall}{\sim} pc', x'$$

iff

$$\begin{array}{l} \forall p \in \text{Paths}(P), p_1, p_2, p_3, \sigma, \chi, \sigma', \chi' \\ \qquad \Big(p = p_1 \cdot \langle pc, \sigma, \chi \rangle \cdot p_2 \cdot \langle pc', \sigma', \chi' \rangle \cdot p_3 \Rightarrow \sigma(x) = \sigma'(x') \Big) \end{array}$$

Two variables at two program points are must-aliases if on every run of the program, whenever they are both reached, the value of the first variable at the first program point is the same as the value of the second variable at the second program point.

To make practical analysis easier and to be able to reuse existing analysis techniques we divided postdominating aliases into two separate parts. But now we must check that they combine to form an approximation of the original definition.

Lemma 6 (Postdomination combined with must-aliasing under-approximates postdominating aliasing)**.**

$$\forall P, pc, x, pc', x' \left(\begin{array}{c} (pc', pc) \in \text{postdoms}(\text{Strip}(\text{Paths}(P))) \\ P \vdash pc, x \overset{\forall}{\sim} pc', x' \end{array} \Rightarrow P \vdash pc', x' \text{ pda } pc, x \right)$$

Proof.

Assume arbitrary P, pc, x, pc', x'.

By lemma 5

$$\forall p \in \text{Paths}(P), p_1, p_2, \sigma, \chi \; (p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \Rightarrow \exists \sigma', \chi'.\ \langle pc', \sigma', \chi'\rangle \in p_2) \tag{4.73}$$

By definition 16

$$\begin{gathered}\forall p \in \text{Paths}(P), p_1, p_2, p_3, \sigma, \chi, \sigma', \chi' \\ \Big(p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_3 \Rightarrow \sigma(x) = \sigma'(x') \Big)\end{gathered} \tag{4.74}$$

Show, from definition 13

$$\forall p \in \text{Paths}(P), \sigma, \chi, p_1, p_2$$
$$\left(p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \;\Rightarrow\; \exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array} \right)$$

Assume arbitrary p, σ, χ, p_1, p_2.

Assume

$$p \in \text{Paths}(P) \tag{4.75}$$

$$p = p_1 \cdot \langle pc, \sigma, \chi\rangle \cdot p_2 \tag{4.76}$$

Show

$$\exists \sigma', \chi', p_3, p_4. \begin{array}{c} p_2 = p_3 \cdot \langle pc', \sigma', \chi'\rangle \cdot p_4 \\ \sigma(x) = \sigma'(x') \end{array}$$

By (4.75), (4.76), and (4.73)

$$\exists \sigma', \chi'.\ \langle pc', \sigma', \chi'\rangle \in p_2 \tag{4.77}$$

By (4.77), (4.76) and (4.74)

$$\exists \sigma', \chi'. \begin{array}{c} \langle pc', \sigma', \chi'\rangle \in p_2 \\ \sigma(x) = \sigma'(x') \end{array}$$

$\square$

Must-aliasing is not possible to establish in general [21] but we can use a restricted approximation, kill analysis, to decide whether the *same variable* refers to the same value at two different points. In practice, the results we achieve with this limited approximation are quite good (chapter 7).

Definition 17 (Kill analysis) NK

$$P \vdash \mathrm{NK}(pc, pc', x)$$

iff

$$\forall \sigma, \chi, \sigma', \chi', \sigma'', \chi'' \left(\begin{array}{c} P \vdash \mathsf{init} \rightsquigarrow^* \langle pc, \sigma, \chi \rangle \\ P \vdash \langle pc, \sigma, \chi \rangle \rightsquigarrow^* \langle pc', \sigma', \chi' \rangle \\ P \vdash \langle pc', \sigma', \chi' \rangle \rightsquigarrow^* \langle pc'', \sigma'', \chi'' \rangle \end{array} \Rightarrow \sigma(x) = \sigma''(x) \right)$$

Lemma 7 (Kill analysis under-approximates must-aliasing)**.**

$$\forall P, pc, pc', x \left(P \vdash \mathrm{NK}(pc, pc', x) \;\Rightarrow\; P \vdash pc, x \overset{\forall}{\sim} pc', x \right)$$

4.8 Control flow graph

Domination, postdomination and must-aliasing are defined over real execution traces so we still need to approximate them further. We will do this using a control flow graph. We only define its signature here as the details depend on the particular language semantics.

Definition 18 (Inter-procedural control flow graph) intercfg

$$\mathrm{intercfg} : \mathbf{Program} \rightarrow \mathcal{P}\left(\mathbf{Counter} \times \mathbf{Counter}\right)$$

We just assume it has been calculated using flow analysis (sections 1.4 and 2.5) and has the usual property that it is an over-approximation of actual execution.

Assumption 2 (Control flow graph over-approximates execution)**.**

$$\forall P, pc, \sigma, \chi, pc', \sigma', \chi' \left(P \vdash \langle pc, \sigma, \chi \rangle \rightsquigarrow \langle pc', \sigma', \chi' \rangle \Rightarrow (pc, pc') \in \mathrm{intercfg}(P) \right)$$

Similarly to definition 4 for actual execution, we define the set of paths for a program based on the transitions in the control flow graph.

Definition 19 (Inter-procedural CFG paths) InterCfgPaths

$$\text{InterCfgPaths} : \mathbf{Program} \to \mathcal{P}\left(\mathbf{Path}_{\mathbf{Counter}}\right)$$

$$\text{InterCfgPaths}(P) = \left\{ p \;\middle|\; \begin{array}{c} p \in \text{ReachingPaths}(P) \\ \exists p', pc.\, p = p' \cdot pc \wedge pc \in \text{counters}(\mathsf{final}) \end{array} \right\}$$

where

$$\text{counters}(cs) = \{ pc \mid \langle pc, \sigma, \chi \rangle \in cs \}$$

$$\text{ReachingPaths}(P) = 1 \cup \left\{ p \cdot pc \cdot pc' \;\middle|\; \begin{array}{c} p \cdot pc \in \text{ReachingPaths}(P) \\ (pc, pc') \in \text{intercfg}(P) \end{array} \right\}$$

As the control flow graph over-approximates execution, the paths created from it over-approximate the actual execution paths.

Lemma 8 (CFG paths over-approximate execution paths)**.**

$$\text{InterCfgPaths}(P) \supseteq \text{Strip}(\text{Paths}(P))$$

Paths and their respective dominations and postdominations have the following property which we use here to show that our conservative over-approximation of real execution paths still renders a conservative under-approximation of real dominators and postdominators.

Lemma 9.

$$\forall Q, P \in \mathcal{P}\left(\mathbf{Path}_N\right) \left(P \subseteq Q \;\Rightarrow\; \text{doms}(Q) \subseteq \text{doms}(P)\right)$$

Lemma 10.

$$\forall Q, P \in \mathcal{P}\left(\mathbf{Path}_N\right) \left(P \subseteq Q \;\Rightarrow\; \text{postdoms}(Q) \subseteq \text{postdoms}(P)\right)$$

The approximation is now computable but to be useful it must be a conservative approximation. When one set of paths over-approximates another—as in the case of our CFG paths over-approximating the real execution paths—the dominator sets under-approximate each other. This is precisely the relationship we need in order to conservatively approximate postdominating aliasing using the control flow graph.

Corollary 1 (CFG dominators under-approximate real dominators)**.**

$$\text{doms}(\text{InterCfgPaths}(P)) \subseteq \text{doms}(\text{Strip}(\text{Paths}(P)))$$

Corollary of lemma 9 and lemma 8.

Corollary 2 (CFG postdominators under-approximate real postdominators)**.**

$$\text{postdoms}(\text{InterCfgPaths}(P)) \subseteq \text{postdoms}(\text{Strip}(\text{Paths}(P)))$$

Corollary of lemma 10 and lemma 8.

4.9 Constrained CFG

The inter-procedural control flow graph in the previous section is computable and conservative (assumption 2) but, in a higher order language, flow analysis is required in order to achieve any degree of precision. In section 2.5 we detailed the challenges of precise flow analysis. Indeed our approach is intended to improve the precision of flow analysis to avoid the expense of the more precise variants. So requiring a precise inter-procedural control flow graph is counter-productive.

And yet our formalism, as defined so far, would appear to do so. The approximations of domination and postdomination are defined in terms of paths through the CFG. An imprecise CFG would lead to many actual dominators and postdominators being excluded by the approximation as call receivers are conservatively estimated to jump execution anywhere. So the question is, can we still approximate them using an intra-procedural control flow graph that no longer conservatively approximates execution? Fortunately we can.

4.9.1 Postdomination on constrained graph remains sound

To prove this, we show that a set of paths constrained to include only a specific subset of the nodes has the same postdomination relation as the original set of paths when the domination relationship is between nodes of the specific subset. If we make the subset be the set of program counters in a particular procedure then any domination relationships within that method continue to be a conservative approximation even though calculating them only takes intra-procedural control flow into account.

First we formalise what it means to constrain a path.

Definition 20 (Partition filter) Constrain

Given N partitioned such that $N = N_1 \uplus N_2$

$$\text{Constrain}_{N_1} \;:\; \mathbf{Path}_N \to \mathbf{Path}_{N_1}$$

$$\text{Constrain}_{N_1}([]) = []$$

$$\text{Constrain}_{N_1}(n \cdot ns) = \begin{cases} n \cdot \text{Constrain}_{N_1}(ns) & \text{if } n \in N_1 \\ \text{Constrain}_{N_1}(ns) & \text{if } n \notin N_1 \end{cases}$$

$$\text{Constrain}_{N_1} \;:\; \mathcal{P}\left(\mathbf{Path}_N\right) \to \mathcal{P}\left(\mathbf{Path}_{N_1}\right)$$

$$\text{Constrain}_{N_1}(ps) = \{\text{Constrain}_{N_1}(p) \mid p \in ps\}$$

The function transforms a path so that it only includes nodes from a partition of the nodes. When the input path joins a node in the partition to one outside it, the function transforms the path so that the node instead joins to the next node in the path that is in the partition again. In this way the function not only filters the path but closes the resulting gaps.

We use the following lemmas in our proofs later. They follow directly from the definition of Constrain.

Lemma 11.

$$\forall N = N_1 \uplus N_2, p \in \mathbf{Path}_N, n \in p \,(n \in N_1 \Rightarrow n \in \text{Constrain}_{N_1}(p))$$

Lemma 12.

$$\begin{aligned}&\forall N = N_1 \uplus N_2, p \cdot p' \in \mathbf{Path}_N \\ &\qquad (\text{Constrain}_{N_1}(p \cdot p') = \text{Constrain}_{N_1}(p) \cdot \text{Constrain}_{N_1}(p'))\end{aligned}$$

Lemma 13.

$$\forall N = N_1 \uplus N_2, p \in \mathbf{Path}_N, n \left(n \in \text{Constrain}_{N_1}(p) \Rightarrow n \in p\right)$$

Lemma 14.

$$\forall N = N_1 \uplus N_2, p \in \mathbf{Path}_N, n$$
$$\left(\begin{array}{l} \text{Constrain}_{N_1}(p) = p_1 \cdot n \cdot p_2 \Rightarrow \\ \qquad\qquad \exists p_3 \cdot n \cdot p_4. \text{Constrain}_{N_1}(p_3) = p_1 \wedge \text{Constrain}_{N_1}(p_4) = p_2 \end{array}\right)$$

Our theory states that a set of constrained paths maintains all the domination relationships of the original paths when between nodes in the contraining set. We only prove this for postdomination; the proof for domination takes the same form.

Theorem 6.

$$\forall N = N_1 \uplus N_2, Q \in \mathcal{P}\left(\mathbf{Path}_N\right)$$
$$\left(\text{postdoms}(\{\text{Constrain}_{N_1}(p) \mid p \in Q\}) = \text{postdoms}(Q) \cap (N_1 \times N_1)\right)$$

Proof.

Take arbitrary N, N_1, N_2 and Q such that

$$N = N_1 \uplus N_2 \tag{4.78}$$

$$Q \in \mathcal{P}\left(\mathbf{Path}_N\right) \tag{4.79}$$

Show

$$\text{postdoms}(\text{Constrain}_{N_1}(Q)) \subseteq \text{postdoms}(Q) \cap (N_1 \times N_1) \tag{4.80}$$

Take arbitrary n, n' such that

$$(n, n') \in \text{postdoms}(\text{Constrain}_{N_1}(Q)) \tag{4.81}$$

Show

$$(n, n') \in \text{postdoms}(Q) \cap (N_1 \times N_1)$$

By (4.81), definition 20 and definition 6

$$(n, n') \in N_1 \times N_1 \tag{4.82}$$

Take arbitrary $p \in Q$ such that

$$\exists p_1, p_2 . p = p_1 \cdot n' \cdot p_2 \tag{4.83}$$

Show

$$n \in p_2$$

By (4.83) and lemma 12

$$\text{Constrain}_{N_1}(p) = \text{Constrain}_{N_1}(p_1) \cdot \text{Constrain}_{N_1}(n') \cdot \text{Constrain}_{N_1}(p_2) \tag{4.84}$$

By (4.84) and (4.82)

$$\text{Constrain}_{N_1}(p) = \text{Constrain}_{N_1}(p_1) \cdot n' \cdot \text{Constrain}_{N_1}(p_2) \tag{4.85}$$

By (4.85) and (4.81)

$$n \in \text{Constrain}_{N_1}(p_2) \tag{4.86}$$

By (4.86) and lemma 13

$$n \in p_2 \tag{4.87}$$

Show $\text{postdoms}(Q) \cap (N_1 \times N_1) \subseteq \text{postdoms}(\text{Constrain}_{N_1}(Q))$.

Take arbitrary n, n' such that

$$(n, n') \in \text{postdoms}(Q) \cap (N_1 \times N_1) \tag{4.88}$$

Show

$$(n, n') \in \text{postdoms}(\text{Constrain}_{N_1}(Q))$$

Take arbitrary $p \in \text{Constrain}_{N_1}(Q)$ such that

$$\exists p_1, p_2 . p = p_1 \cdot n' \cdot p_2 \tag{4.89}$$

Show

$$n \in p_1$$

By definition

$$\exists p' . p' \in Q \wedge p = \text{Constrain}_{N_1}(p') \tag{4.90}$$

By (4.90) and lemma 14

$$\exists p_3, p_4 . p' = p_3 \cdot n' \cdot p_4 \tag{4.91}$$

$$\text{Constrain}_{N_1}(p_3) = p_1 \wedge \text{Constrain}_{N_1}(p_4) = p_2 \tag{4.92}$$

By (4.91), (4.88) and definition 6

$$n \in p_4 \tag{4.93}$$

By (4.93), (4.92) and lemma 11

$$n \in p_2 \tag{4.94}$$

□

We define the set of control flow graphs paths within a procedure as the constrained version of the paths that include inter-procedural control flow

---

Definition 21 (Intra-procedural path approximation) IntraCfgPaths

---

$$\text{IntraCfgPaths}(P, proc) = \text{Constrain}_{\{pc \mid pc \in proc\}}(\text{InterCfgPaths}(P))$$

---

and prove that postdominators calculated from these paths conservatively approximate real postdomination.

Theorem 7.

Assuming some function procedures(P) that ranges over the procedures defined in the argument

$$\forall P, \mathit{proc} \in \text{procedures}(P)(\\ \text{postdoms}(\text{IntraCfgPaths}(P, \mathit{proc})) \subseteq \text{postdoms}(\text{Paths}(P)))$$

Proof.

Assume arbitrary P and proc.

By theorem 6 and definition 21

$$\begin{aligned}&\text{postdoms}(\text{IntraCfgPaths}(P, \mathit{proc})) = \\ &\quad \text{postdoms}(\text{Constrain}_{\{pc \mid pc \in \mathit{proc}\}}(\text{InterCfgPaths}(P))) \\ &\qquad \cap \{(pc, pc') \mid pc \in \mathit{proc}, pc' \in \mathit{proc}\}\end{aligned} \tag{4.95}$$

By (4.95)

$$\begin{aligned}&\text{postdoms}(\text{IntraCfgPaths}(P, \mathit{proc})) \subseteq \\ &\quad \text{postdoms}(\text{Constrain}_{\{pc \mid pc \in \mathit{proc}\}}(\text{InterCfgPaths}(P)))\end{aligned} \tag{4.96}$$

By (4.96) and corollary 2

$$\text{postdoms}(\text{IntraCfgPaths}(P, \mathit{proc})) \subseteq \text{postdoms}(\text{Strip}(\text{Paths}(P)))$$

□

We have defined a computable approximation of postdomination based both and inter-procedural and an intra-procedural control flow graph. We have also defined an approximation of must-aliases through kill analysis.

As a quick sanity check we prove the following theorem that the various approximations combine to form an approximation of required features.

Theorem 8 (NK and postdoms(IntraCfgPaths(P, proc)) approximates $\vdash_{\text{RF}}$)**.**

$$\forall P, proc \in \text{procedures}(P), pc, pc', x, m$$
$$\left(\begin{array}{c}P, pc' \vdash_{\text{RF}} x : m\\ pc \in proc\\ pc' \in proc\\ (pc', pc) \in \text{postdoms}(\text{IntraCfgPaths}(P, proc))\\ P \vdash \text{NK}(pc, pc', x)\end{array} \Rightarrow P, pc \vdash_{\text{RF}} x : m\right)$$

Proof.

Assume arbitrary P, $proc$, pc, pc', x and m.

Given

$$P, pc' \vdash_{\text{RF}} x : m \tag{4.97}$$
$$pc \in proc \tag{4.98}$$
$$pc' \in proc \tag{4.99}$$
$$(pc', pc) \in \text{postdoms}(\text{IntraCfgPaths}(P, proc)) \tag{4.100}$$
$$P \vdash \text{NK}(pc, pc', x) \tag{4.101}$$

Show that

$$P, pc \vdash_{\text{RF}} x : m$$

By lemma 7 and (4.101)

$$P \vdash pc', x \overset{\forall}{\sim} pc, x \tag{4.102}$$

By theorem 7 and (4.100)

$$(pc', pc) \in \text{postdoms}(\text{InterCfgPaths}(P)) \tag{4.103}$$

By corollary 2 and (4.103)

$$(pc', pc) \in \text{postdoms}(\text{Strip}(\text{Paths}(P))) \tag{4.104}$$

By lemma 6, (4.104) and (4.102)

$$P \vdash pc', x \text{ pda } pc, x \tag{4.105}$$

By theorem 4, (4.97) and (4.105)

$$P, pc \vdash_{\text{RF}} x : m \tag{4.106}$$

□

All that now remains for a computable approximation of dominating and postdominating aliasing, and therefore observed and required features, is computable approximation of the kill analysis. This is dependent on the particular language so now we define one, loosely based on Python, as a case study.

4.10 Case study: Imperative, object-oriented language

In this section we define a language that, though based on Python, is *idealised*. By this we mean that it includes all of the features needed to allow our approach to function and none of the features that would prevent it (section 3.5). Although some 'pathologically dynamic' features have been excluded, the language is still dynamically typed in the duck typed manner that Python is so well known for. It remains similar enough to Python to model the cases that make type inference difficult in a dynamically typed language (section 1.3). Similarities include:

- duck typed semantics
- class based value creation
- values are objects
- variables are mutable
- execution is input-dependent

The most obvious difference is the syntax which is three-address form. As such, the language can be considered an intermediate language into which

Python programs are translated. Three-address form is convenient as we can reason directly using the framework laid out in this chapter so far.

The most important differences are that the features of an object are fixed and, in particular, are dictated by classes which are immutable. In Python classes are first-class, mutable objects and features can be added and removed freely from the instances they create. When we come to evaluate our implementation on fully-fledged Python (chapter 7) we show that ignoring this difference does not fatally harm the result.

Another notable difference is the absence of exceptions. While exceptions are just a control flow mechanism and therefore would appear as extra transitions in the control flow graph, an intra-procedural analysis would have to approximate the transitions conservatively meaning that every statement could throw an exception and that exception could flow control anywhere. The result is that no program point would ever be a postdominator rendering our analysis impotent. In practice (section 5.5) we ignore exceptions as one would have to be used for polymorphic path selection before ignoring it could harm our result. We believe this to be rare. Our evaluation of our implementation on fully-fledged Python found no cases where such a use of exceptions affected the result.

A summary of some of the differences between Python and our toy language are:

- our classes are immutable
- our objects' features are fixed
- our language has no facility for runtime code generation (eval etc.)
- our language does not permit reflection
- our language does not have exceptions
- all our code is in methods

We adjust and extend the base sets, S, defined in section 4.2, with the sets S^{π}, to include the specifics of the language.

Figure 4.7: Toy language syntax

trm ::=		terms:
	$x, y, self$	variable
	$x.m(y)$	method call
	$n()$	construction
	read	input
$stmt$ ::=		statements:
	$x = trm$	assignment
	if x: pc	conditional
	return x	method return
cls ::=		classes:
	class n: $\overline{meth}$	
$meth$ ::=		methods:
	def $m(self, x)$: $\overline{stmt}$ **return** x	

$\boxed{\mathsf{S}^{\pi}}$

$stmt \in \textbf{Statement}$	statements
$\textbf{MethodBody} = \textbf{Counter} \to \textbf{Statement}$	method bodies
$\textbf{Class} = \textbf{Feature} \to \textbf{MethodBody}$	classes
$\nu \in \textbf{Value} = \textbf{Feature} \to \textbf{MethodBody}$	objects
$P \in \textbf{Program} = \textbf{ClassId} \to \textbf{Class}$	programs

The heap is assumed to be pre-populated with three special values, ι_F, ι_T and ι_N, that represent False, True and None, respectively. These values have significance in the conditional statement (E-IfTrue and E-IfFalse in figure 4.8). There is only one configuration, c_ω, in TROUBLE, and all failing duck tests move the program to this configuration.

Definition 22 (Execution, cont.) $\rightsquigarrow$

$$\frac{P(pc) \equiv x = y \qquad \sigma \equiv \sigma'.\varphi \qquad \sigma'' \equiv \sigma'.\varphi[x \mapsto \varphi(y)]}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc+1, \sigma'', \chi\rangle}\ \textsc{E-Var}$$

$$\frac{\begin{array}{c}P(pc) \equiv z = x.m(y) \qquad \iota \equiv \sigma(x) \qquad \mathrm{dom}(\chi(\iota)) \ni m \\ \sigma' \equiv \sigma.\varphi \qquad \varphi \equiv (self \mapsto \iota, x \mapsto \sigma(y), base \mapsto pc) \qquad pc \equiv \chi(\iota)(m)()\end{array}}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc', \sigma', \chi\rangle}\ \textsc{E-Call}$$

$$\frac{P(pc) \equiv z = x.m(y) \qquad m \notin \mathrm{dom}(\chi(\sigma(x)))}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow c_\omega}\ \textsc{E-CallTrouble}$$

$$\frac{\begin{array}{c}P(pc) \equiv x = n() \qquad \iota \notin \mathrm{dom}(\chi) \qquad n \equiv \{\overline{m}\} \\ \chi' \equiv \chi[\iota \mapsto \{\overline{m}\}] \qquad \sigma \equiv \sigma'.\varphi \qquad \sigma'' \equiv \sigma'.\varphi[x \mapsto \iota]\end{array}}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc+1, \sigma'', \chi\rangle}\ \textsc{E-Constructor}$$

$$\frac{P(pc) \equiv x = \textbf{read} \qquad \sigma \equiv \sigma'.\varphi \qquad \iota \equiv \text{arbitrary} \qquad \sigma'' \equiv \sigma'.\varphi[x \mapsto \iota]}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc+1, \sigma'', \chi\rangle}\ \textsc{E-Read}$$

$$\frac{P(pc) \equiv \textbf{if}\ x\colon pc_t \qquad \sigma(x) \neq \iota_F \qquad \sigma(x) \neq \iota_N}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc_t, \sigma, \chi\rangle}\ \textsc{E-IfTrue}$$

$$\frac{P(pc) \equiv \textbf{if}\ x\colon pc_t \qquad \sigma(x) \equiv \iota_F \vee \sigma(x) \equiv \iota_N}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc+1, \sigma, \chi\rangle}\ \textsc{E-IfFalse}$$

$$\frac{\begin{array}{c}P(pc) \equiv \textbf{return}\, x \qquad \sigma \equiv \sigma'.\varphi'.\varphi \\ pc' \equiv \varphi(base) \qquad P(pc') \equiv z = v.m(u) \qquad \sigma'' \equiv \sigma'.\varphi'[z \mapsto \sigma(x)]\end{array}}{P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow \langle pc'+1, \sigma'', \chi\rangle}\ \textsc{E-Return}$$

Figure 4.8: Operational semantics for toy language

4.10.1 CFG-based kill analysis

In our language, with mutable variables, we can approximate an alias analysis by looking at statements that modify the stack. Kill analysis can be performed intra-procedurally by establishing the absence of various mutating statements.

Definition 23 (Intra-procedural kill analysis) IntraNK

$$P \vdash \text{IntraNK}(pc, pc', x)$$

iff

$$\exists proc \in \text{procedures}(P).\forall p, p_1, p_2, p_3, pc''.$$

$$\begin{array}{c} pc \in proc \\ pc' \in proc \end{array} \wedge \left(\begin{array}{c} p \in \text{IntraCfgPaths}(P, proc) \\ p = p_1 \cdot pc \cdot p_2 \cdot pc' \cdot p_3 \\ pc'' \in pc \cdot p_2 \end{array} \Rightarrow \begin{array}{r} \forall y.P(pc'') \neq x = y \\ \forall n.P(pc'') \neq x = n() \\ P(pc'') \neq x = \textbf{read} \\ \forall v, w, m.P(pc'') \neq x = v.m(w) \end{array} \right)$$

Definition 23 ensures that two appearances of a variable only must-alias if they appear in the same procedure, and no *intra-procedural* control-flow path exists that includes any of several prohibited statements. The prohibited statements are those that, according to their operational semantics, can modify the top stack frame (figure 4.8). The conditions in definition 23 only prohibit them if they target the variable whose must-aliasing is in question. For variable assignment (E-Var), constructor assignment (E-Constructor) and input (E-Read) it is obvious from the semantics that the modification to the top stack frame is limited to the target variable's value so it is safe to allow intervening assignments to unrelated variables.

Calls (E-Call) insert a completely new stack frame but, from the syntax (figure 4.7) we know that every call has a matching return statement (E-Return) which restores the original top of the stack with the exception that it may change that frame's value for the call return value target. Therefore we can treat calls in the same way as the other assignments.

Although return statements change the stack, we do not include them in the prohibited statements as any intra-procedural path is guaranteed to end in a return statement and so cannot appear before *pc'* as required in definition 23.

Lemma 15 (IntraNK under-approximates NK)**.**

$$\forall P, pc, pc', x\,(P \vdash \text{IntraNK}(pc, pc', x) \;\Rightarrow\; P \vdash \text{NK}(pc, pc', x))$$

4.10.2 We have a reasonable approximation

In the previous sections we have approximated the definition of required features in stages by breaking it into pieces, each of which has a computable approximation. Most of these pieces are independent of a particular programming language, but approximating must-aliasing requires us to define a specific language which we used to approximate that last piece.

Now we show that these separate, computable approximations combine to produce a sound approximation of our goal, required features. First, we must have some way to obtain the basic set of required features directly from the syntax. We show that this is the case for our toy language from its operational semantics.

Lemma 16 (Some duck tests are syntactically apparent)**.**

$$\forall P, pc, x, m\,(\exists u, v.\, P(pc) \equiv u = x.m(v) \;\Rightarrow\; P, pc \vdash_{\mathrm{SDT}} x : m)$$

Proof.

Take P, pc, x, m arbitrary.

Given

$$\exists u, v.\, P(pc) \equiv u = x.m(v) \tag{4.107}$$

Show

$$\forall \sigma, \chi\,(P \vdash \mathsf{init} \rightsquigarrow^* \langle pc, \sigma, \chi\rangle \Rightarrow P, pc, \sigma \vdash_{\mathrm{DT}} \sigma(x) : m)$$

Take σ, χ arbitrary.

Assume

$$P \vdash \mathsf{init} \rightsquigarrow^* \langle pc, \sigma, \chi\rangle \tag{4.108}$$

Show

$$pc \notin \{pc_f \mid \langle pc_f, \sigma, \chi\rangle \in \mathsf{final}\}$$
$$\forall \chi\,(m \notin \mathrm{dom}(\chi(\sigma(x))) \Rightarrow \forall c\,(P \vdash \langle pc, \sigma, \chi\rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE}))$$

Take χ' arbitrary.

Assume

$$m \notin \mathrm{dom}(\chi'(\sigma(x))) \tag{4.109}$$

Show

$$\forall c\,(P \vdash \langle pc, \sigma, \chi' \rangle \rightsquigarrow c \Rightarrow c \in \mathsf{TROUBLE})$$

Take c arbitrary.

Assume

$$P \vdash \langle pc, \sigma, \chi' \rangle \rightsquigarrow c \tag{4.110}$$

Show

$$c \in \mathsf{TROUBLE}$$

By (4.110), (4.109), (4.107) and definition 22

$$c = c_\omega$$

By (4.107) and definition 22

$$pc \notin \{pc_f \mid \langle pc_f, \sigma, \chi \rangle \in \mathsf{final}\}$$

$\square$

Finally we put all the parts together to approximate required and observed features on an intra-procedural control-flow graph in our toy language.

Required features

Theorem 9 (We can approximate required features)**.**

$$\forall P, proc \in \text{procedures}(P), pc, pc', x, m$$
$$\left(\begin{array}{c} \exists u, v.\, P(pc') \equiv u = x.m(v) \\ pc \in proc \\ pc' \in proc \\ (pc', pc) \in \text{postdoms}(\text{IntraCfgPaths}(P, proc)) \\ P \vdash \text{IntraNK}(pc, pc', x) \end{array} \Rightarrow P, pc \vdash_{\text{RF}} x : m\right)$$

Proof.

Assume arbitrary P, pc, pc', x and m.

Given

$$\exists u, v.\, P(pc') \equiv u = x.m(v) \tag{4.111}$$

$$pc \in proc \tag{4.112}$$

$$pc' \in proc \tag{4.113}$$

$$(pc', pc) \in \text{postdoms}(\text{IntraCfgPaths}(P, proc)) \tag{4.114}$$

$$P \vdash \text{IntraNK}(pc, pc', x) \tag{4.115}$$

Show that

$$P, pc \vdash_{\text{RF}} x : m$$

By lemma 15 and (4.115)

$$P \vdash \text{NK}(pc, pc', x) \tag{4.116}$$

By (4.111) and lemma 16

$$P, pc' \vdash_{\text{SDT}} x : m \tag{4.117}$$

By (4.117) and lemma 3

$$P, pc' \vdash_{\text{RF}} x : m \tag{4.118}$$

By theorem 8, (4.118), (4.112)–(4.114) and (4.116)

$$P, pc \vdash_{\text{RF}} x : m \tag{4.119}$$

□

We have shown that a computable, sound approximation of required features exists for a language and that, for well formed programs written in that language, the variables are guaranteed to have the features required by the approximation.

4.11 Contraindication

Our motivation for inferring observed and required features is to use them to improve the precision of the concrete types inferred using an inter-procedural flow analysis, a technique we have called *contraindication.*

A concrete type is set of abstract values for an expression. Concrete types are sound if they guarantee that expressions only evaluate to values described by the abstract values in the type. Exactly what form these abstract values take varies according to the particular language semantics. In the sets S^{+}, defined in section 4.2, we called these abstract values ‘classes’. What the class interfaces range over is language-dependent and left undefined for now. For an object-oriented language like our toy language (section 4.10) they might map method names to method bodies.

In our formalism, we decide whether a concrete type describes a value based on whether the value’s interface matches that of any of the classes. This makes it straightforward to contraindicate members of the concrete type based on the observed and required features inferred for the expression.

Definition 24 (Valid concrete type) $\models_{\mathrm{C}}$

$$P, pc \models_{\mathrm{C}} x : t$$

iff

$$\forall \sigma, \chi \, (P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi \rangle \;\Rightarrow\; \exists n \in t.\ \mathrm{dom}(n) = \mathrm{dom}(\chi(\sigma(x))))$$

We use concrete types from flow-based type inference as the types to refine using contraindication. We do not define a flow analysis in this work. We assume such an analysis exists for the language in question.

Definition 25 (Flow-based type inference) $\vdash_{\mathrm{CFA}}$

$$P, pc \vdash_{\mathrm{CFA}} x : t$$

As described in section 2.5 there are many flow analysis variations with different trade-offs, some more suitable for some languages than other. We just assume that any such analysis infers sound concrete types.

Assumption 3 (Flow-based type inference produces sound types)**.**

$$\forall P, pc, x, t\, (P, pc \vdash_{\text{CFA}} x : t \;\Rightarrow\; P, pc \models_{\text{C}} x : t)$$

We also assume the semantics of the language are such that a value's interface—its set of features—is not allowed to diverge from that dictated by its creation class. This ensures that any value at the expression supports at least the intersection of the feature defined by each member of the concrete type. This is what enables contraindication to bound the concrete type using the interface inferred from the duck tests. Contraindication is only sound for languages where a value's features are fixed in this way.

Definition 26 (Contraindicated flow type) $\vdash_{\text{CI}}$

$$P, pc \vdash_{\text{CI}} x : t$$

iff

$$\exists t'. \begin{array}{c} P, pc \vdash_{\text{CFA}} x : t' \\ t = \{n \mid n \in t', \forall m\, (P, pc \models_{\text{F}} x : m \;\Rightarrow\; m \in \text{dom}(n))\} \end{array}$$

Theorem 10 (Contraindicated types remain sound)**.**

$$\forall P, pc, x, t\, \Big(P, pc \vdash_{\text{CI}} x : t \Rightarrow P, pc \models_{\text{C}} x : t\Big)$$

Proof.

Given

$$P, pc \vdash_{\text{CI}} x : t \tag{4.120}$$

Show

$$P, pc \models_{\text{C}} x : t$$

By (4.120) and definition 26

$$\exists t'. \begin{array}{c} P, pc \vdash_{\mathrm{CFA}} x : t' \\ t = \{n \mid n \in t', \forall m\,(P, pc \models_{\mathrm{F}} x : m \;\Rightarrow\; m \in \mathrm{dom}(n))\,\} \end{array} \tag{4.121}$$

Show

$$\forall \sigma, \chi\,(P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi \rangle \;\Rightarrow\; \exists n \in t.\ \mathrm{dom}(n) = \mathrm{dom}(\chi(\sigma(x))))$$

Assume arbitrary σ, χ and

$$P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi \rangle \tag{4.122}$$

Show

$$\exists n \in t.\ \mathrm{dom}(n) = \mathrm{dom}(\chi(\sigma(x)))$$

Assume t' such that

$$P, pc \vdash_{\mathrm{CFA}} x : t' \tag{4.123}$$

$$t = \{n \mid n \in t', \forall m\,(P, pc \models_{\mathrm{F}} x : m \;\Rightarrow\; m \in \mathrm{dom}(n))\,\} \tag{4.124}$$

By (4.123), (4.122) and assumption 3

$$\exists n \in t'.\ \mathrm{dom}(n) = \mathrm{dom}(\sigma(\chi(x))) \tag{4.125}$$

By (4.125), take n such that

$$n \in t' \tag{4.126}$$

$$\mathrm{dom}(n) = \mathrm{dom}(\chi(\sigma(x))) \tag{4.127}$$

By (4.124) and (4.126)

$$\forall m\,(P, pc \models_{\mathrm{F}} x : m \Rightarrow m \in \mathrm{dom}(n)) \;\Rightarrow\; n \in t \tag{4.128}$$

Take m arbitrary.

Assume

$$P, pc \models_{\mathrm{F}} x : m \tag{4.129}$$

Show

$$m \in \mathrm{dom}(n)$$

By (4.129) and definition 10

$$\forall \sigma, \chi\, (P \vdash \mathsf{init} \rightsquigarrow^{*} \langle pc, \sigma, \chi \rangle \;\Rightarrow\; m \in \mathrm{dom}(\chi(\sigma(x)))) \tag{4.130}$$

By (4.130) and (4.122)

$$m \in \mathrm{dom}(\chi(\sigma(x))) \tag{4.131}$$

By (4.131) and (4.127)

$$m \in \mathrm{dom}(n) \tag{4.132}$$

By (4.129)–(4.132)

$$\forall m\, (P, pc \models_{\mathrm{F}} x : m \Rightarrow m \in \mathrm{dom}(n)) \tag{4.133}$$

By (4.133) and (4.128)

$$n \in t \tag{4.134}$$

By (4.132) and (4.127)

$$\exists n \in t.\ \mathrm{dom}(n) = \mathrm{dom}(\chi(\sigma(x)))$$

$\square$

Theorem 10 means that contraindication safely refines concrete types as long as those types are, themselves, sound. Although contraindication is defined in terms of the actual features of a variable, which is undecidable, we can under approximate these by approximating observed and required features and assuming a well formed program.

4.12 Prerequisites

In section 4.10 we applied our formalism to a Python-inspired toy language and in the next chapter we explore the challenges of mapping those concepts onto Python proper. But before we do so, we revisit the general, language independent concepts and summarise the prerequisites that make them applicable. This is important because, although we have only applied our techniques to object oriented languages so far, we designed our technique to avoid *depending* on object orientation in any way. We believe that the techniques apply to other language paradigms equally well as long as the language meets the prerequisites (section 8.5).

Approximating required features Required features have applications outside contraindication; Lindahl and Sagonas [23] use something similar in their success types for type checking. Therefore we take care to separate the prerequisites needed to approximate required features from the prerequisites needed to use required features for contraindication. It is worth noting that these prerequisites are needed for our approximation of required features that starts with a basic set and propagates its members. Other approximations may have different prerequisites.

Trouble Some definition of *detectable failure*, TROUBLE, that, should it occur, indicates an ill-formed program.

TROUBLE does *not* necessarily have to halt the program. Despite the name, 'required feature' does not mean the feature is actually required in any absolute sense. It is only required with respect to the current definition of TROUBLE chosen to suit the application at hand. The promise a required feature makes about the consequence of a missing feature is that TROUBLE occurs, nothing more. Then it is up to the application of required features to decide whether TROUBLE implies halting.

For a hint at how general we think this might be, see the discussion on squeaking in section 8.2.

Basic set Some *basic set* of required features inferred from *syntactically evident* duck tests; syntax that the language semantics guarantees will cause the program to encounter TROUBLE if the required feature is missing.

Fixed features If the basic set is to be propagated further, the features supported by value must be fixed throughout its lifetime. Were this not the case a feature could be required in the basic set after it had been added to the value but propagated to a point before it was added where it is no longer required.

Postdomination It must be possible to calculate a conservative under-approximation of the postdominators of a program point.

Must-aliasing It must be possible to calculate a conservative under-approximation of which variables must-alias which others. This may be very restricted, as in the later parts of our formalism where the relation is restricted to instances of the same variable name within a single procedure.

Observed features have the same prerequisites except that postdomination becomes domination.

Contraindication Contraindication uses required features so all the prerequisites above apply in addition to a few others.

Upper bound Contraindication narrows down the possible classes of value at a variable using the features any such value must have. But in order to perform the narrowing there must be a way to detect that a class and a set of features are incompatible. As a general prerequisite, that assumes little about the nature of classes and nothing about features, we just say that the classes must define the *upper bound* of the features any of their values can support. A set of features that is not a subset of this upper bound is incompatible with that class and the class can be excluded from the type.

Well formed Contraindication assumes a well formed program which means one that will not lead to TROUBLE. The contraindicated type is only sound under this assumption. Exactly what well formed means, however, depends on the definition of TROUBLE. An obvious definition is to make TROUBLE a halting type error state, making well formed a promised that the program will not halt in an type error state. However there may be situations where some other definition is more convenient or, as in the case of Python in chapter 5, required by the semantics of the language. What is important is that whatever definition of TROUBLE is chosen, it must be reasonable in that situation for the programmer to be able to promise it does not occur.

Flow analysis Contraindication requires concrete types, in the form of sets of classes, that we can compare the sets of required features against. Furthermore, the concrete types must be sound. This generally requires some form of flow-analysis-based type inference. We do not mandate any particular level of precision. A trivially imprecise analysis could just infer that all values are instances of some class in the system. This amounts to using *only* required features to infer types.

All the analyses including the flow analyses have a further prerequisite

Closed world As is common in the dynamically typed sphere where static type annotations cannot not act as partial result stores, the analyses rely on having *all* code that makes up the running system available for analysis.

5 Practical language

In section 4.10 we use an idealised duck-typed toy language to demonstrate the concepts of our formalism. This language has all the properties necessary to make our technique sound and, just as importantly, it omits any properties that make it unsound. In this chapter we explore the challenges and compromises of mapping the concepts of chapter 4 onto a real-world language.

5.1 Python

Python, unlike our Python-inspired toy language from section 4.10, is a mature, practical, duck-typed language rich with features that make the design of semantically meaningful development tools difficult. Whereas we could tailor the features of our toy language precisely to match our prerequisites (section 4.12), Python's features developed organically over time with no regard for static analysis. While complicating our task, this also makes it precisely the kind of language most likely to benefit from such work.

There are disparities between the guarantees our approach requires and the guarantees Python provides, but all is not lost. Python may provide rich, and sometimes pathologically dynamic, features but that does not mean people use them [9]. Even if they do—studies disagree [19]—our results show they do not use such features in a way that coordinates polymorphism, leaving our inferred types largely intact. Furthermore, most of the important concepts can be mapped directly with a little care. We base our toy language on Python precisely to explore this mapping.

Although the toy language in section 4.10 uses an intermediate three-address form with only variables having required features, Python can have arbitrarily nested expressions, each with their own required features. Mapping concepts between the two is not a problem, however, as such expressions can be converted to three-address form using temporary variables. For example,

```
func(x.y()).m()
```

becomes

```
temp_1 = x.y()
temp_2 = func(temp_1)
temp_2.m()
```

In the rest of our discussion, as in the formalism, we talk about variables on the understanding that this might mean an expression that has been converted in this way.

5.2 Defining TROUBLE

Deciding on an acceptable definition of TROUBLE is subtly difficult. The definition is intertwined with others:

- Whatever the definition, it must allow *syntactically evident* duck tests to be inferred directly from the syntax; the language semantics must guarantee TROUBLE should the value referenced at the syntactic element not support the required feature.
- Well-formed programs are defined in terms of TROUBLE: any program that can never encounter TROUBLE is well formed; all others are ill formed. Whatever the definition of TROUBLE, such an assumption needs to be sensible.

In the semantics of toy Python (figure 4.8) TROUBLE is a halting state entered if a program attempts to call a method that a value does not possess. Real Python has similar semantics: accessing a method or field that does not exist in the object raises an AttributeError exception. However, there is an important difference between the toy language and Python: in Python the program can catch the exception in an exception handler and continue execution.

Assuming we want to use the presence of method calls in the syntax to produce our basic set—we do—our definition of TROUBLE is that an AttributeError was raised. A well-formed program is now defined as one that will never raise such an error. The question becomes, is it reasonable for contraindication to assume a program does not raise an AttributeError ?

Let us first assume the halting case where the exception is not caught and propagates to the top of the program, halting it with a run-time type error. We argue that it is entirely reasonable to assume an input program does not allow this to happen. Any mature project must have all but extinguished the possibility of run-time type errors halting the program. Even during development, any sensible program will be largely type correct, not to mention any libraries it depends on, which includes the system libraries. The assumption that the input program be type correct is not unique to our approach, in fact it is almost universal in program analysis tools. Even in statically typed languages, development tools do not promise to give correct results if the input program does not pass the static type checker, a high bar.

5.2.1 No reflection

But we are not just dealing with a halting case: an AttributeError can be caught and handled like so:

```
1  x.n()
2  try:
3      x.m()
4  except AttributeError:
5      pass
```

This is a form of reflection or introspection, and amounts to a trial-and-error attempt to use a feature of a value and trap then handle the run-time type error if it fails. But we have to assume the input program never raises an AttributeError , regardless of any subsequent handling. If the code, nevertheless, did use this kind of reflection, it could lead to unsound contraindication types as classes are excluded on the basis of methods that are not really present.

The problem occurs because the unsupported feature is actually required; the input program was ill formed by our definition of TROUBLE. Assume that x supports n but not m: the analysis determines x.m() is required at x_3 and, as x_3 is a postdominating alias of x_1, that feature m is also required at x_1.

Contraindication, assuming that the program is in fact well formed, uses theorem 1 to decide that any value at x_1 must support feature m and, as a result, removes any class without that feature from the type inferred by flow analysis. But it is wrong, because values without m can arrive at x_1, just not in well-formed programs.

Whether it is sensible to assume that this behaviour does not occur is more subjective and this remains one of the weaknesses of contraindication when applied to Python. We argue that failing to guarantee correct results in the presence of reflection is not unique to our approach. Moreover, even though this behaviour does occur in real programs, our results indicate that it does not affect the correctness of the contraindicated types in practice (chapter 7).

5.3 The basic set

The basic set (see sections 4.10.2 and 4.12) comes from syntactically evident operations that must agree with our definition of TROUBLE: they guarantee TROUBLE if the value is missing the feature. In Python this corresponds to the syntax for reading from an attribute: an expression followed by a dot and then the name of the attribute without any assignment on the right-hand side. For example:

```
expression.feature_name
```

Looking back at our definition of syntactically evident (section 3.10.1) the expression, expression, is the syntactic element being typed and the attribute, feature_name, is the feature being used. Python's semantics guarantee that attempting to access an attribute will raise an AttributeError if it does not exist in the object[1], which is the property we need for a required feature.

At this point we face a problem because another prerequisite is that a value's features are fixed for its lifetime. However, in Python, non-method attributes—hereafter called fields—can be added to objects at any time. We

[1]Python hackers may be shaking their heads at this point. There are ways to hook arbitrary requests for an attribute and implement custom behaviour that, if the attribute is missing, may choose to do something other than raise an AttributeError. We consider this one of Python's *pathologically dynamic* features akin to modifying the object's interface on the fly, a behaviour we forbid entirely (see section 5.4).

work around the problem in the next section by restricting our basic set as best we can to include only method attributes and not fields.

5.4 Fixed features

Propagating the basic set to other program points relies on the set of features supported by a value—its interface—to be fixed throughout its lifetime. Otherwise a required feature might be propagated from a point where it is required to a point before the feature has been added to the interface, a place where the feature is no longer required.

Our toy language was ideal and its semantics did not provide a way to modify a value's set of methods—the only type of feature a value could possess. Python is not ideal. In particular, fields, a type of feature, can *only* be added after the object is created. Python classes do not have a way to declare that their instances have a particular field. Instead, they spring into existence when they are assigned to, often, but not only, in the constructor. The following example demonstrates why this is a problem.

```
1  x = X()
2  # is 'a' required in 'x' here?
3  x.a = 7
4  print x.a
```

The field access on line 4 would make a a required feature of x_4 in the basic set. The statement on line 4 postdominates the other statements and x_4 must-aliases the other instance of x, so the analysis propagates a as a required feature of x to the other statements. However, the field was added at line 3, so is not, in fact, required earlier.

Our solution is to exclude fields from the basic set and only include methods because methods cannot be added to an object after it is created.[2] Doing so requires us to compromise the definition of syntactically evident slightly.

[2]One again, Python hackers are frowning. Technically it *is* possible to change the methods of an already created object, a technique delightfully known as either Monkey Patching or Duck Punching (`http://www.ericdelabar.com/2008/05/metaprogramming-javascript.html`). However, doing so requires using metaclass manipulation (`http://stackoverflow.com/a/2982/67013`), is exceedingly rare, and falls into the category of pathologically dynamic behaviours that we do not attempt to model.

We filter methods from fields by whether the attribute is immediately called or not. In other words, if this is taken to be a method:

```
e.m()
```

and we add it to the basic set as e requiring feature 'method m' while this is taken to be a field e.f, we ignore it. But this distinction is not perfect because the two categories are not as separate as they appear here.

Sometimes the field syntax is actually used to access a method. Methods are first-class objects in Python so a program can read a method as a callable closure, without calling it, using the field syntax:

```
v = x.m
v() # calls method m
```

Method m is still required by variable x but, because we use calls to distinguish fields from methods, we miss this one. However, as we show in chapter 4, it is safe to under-approximate required features and, in general, any sound analysis will be an under-approximation.

More seriously, sometimes the method syntax is actually used to access a field. Many objects in Python, including first-class functions, are callable. A callable object is just an object and can be assigned to a field of another object. If this object is subsequently called directly from the attribute, the syntax is identical to the method call:

```
def f():
    print "Hello World!"

x = X()
x.m = f
x.f()
```

This is a more serious problem, as it causes us to *over-estimate* the required features with the potential for unsound contraindication. We have not yet found a solution to this problem.

Both issues are caused because we compromised the meaning of syntactically evident when we filtered fields. The definition in section 3.10.1 requires that the syntax uniquely identify the feature being used but, now that we have separated fields and methods into two separate categories of feature, the syntax is supposed to distinguish the categories uniquely. A *method* a should be syntactically distinguishable from a *field* a. We achieve something very close to this ideal using the presence of a call to distinguish the cases but, as we can see, there is some overlap. If the language had truly separate field and method attribute syntax, the problem would not occur.

5.5 Postdomination

Postdomination should be simple to under-approximate; in section 4.9.1 we show how postdomination calculated even on an intra-procedural control flow graph remained a conservative under-approximation. We could just construct such a graph for each Python procedure and apply a standard postdomination algorithm, except that this algorithm would not find any postdominators.

The problems is that the semantics of Python allow any statement to raise an exception. For example, the interpreter signals the system has run out of memory with a MemoryError. This is not a deterministic event, so a control flow graph that *truly* over-approximates execution must include a transition from every statement to the nearest exception handler (which may be the end of the program) in addition to the statement's other transitions. This extra transition prevents postdomination.

Even if we exclude such interpreter exceptions from the control flow graph and only consider exceptions raised with an explicit **raise** statement, we still have a problem of what to do at call sites. We want to use an intra-procedural control flow graph, but doing so means we cannot tell if a given call site might call code that raises an exception—there are no exception specifications in Python.

We argue that it is reasonable to ignore exceptions in our analysis. For an exception to affect our contraindicated type it would coordinate polymorphism. By that we mean that the programmer has to have used exceptions intentionally as a control flow mechanism to direct polymorphic types along different paths through. One obvious example is the reflective behaviour we have already forbidden (section 5.2.1).

But other situations may be more subtle:

```
def feed_rabbit(a):
    if a.favourite_food == "Rabbit":
        a.eat("Rabbit")
```

```
45      else:
46          raise IncompatibleMealError()
47
48  if random():
49      pet = Dog()
50  else:
51      pet = Chicken()
52
53  pet.stroke()
54  feed_rabbit(pet)
55  pet.wag_tail()
```

Similar to the example from section 1.3 and like many subtle duck-typing examples, the programmer has relied on domain-specific knowledge to guide polymorphism such that the program is type correct at run time but in a way that makes static analysis challenging. If our control flow graph ignores exceptions raised inside calls, wag_tail appears to be required at lines 54 and 53 because line 54 postdominates both of them while pet is not killed. But wag_tail is not required when pet is a chicken because the programmer knows that trying to feed their pets a rabbit will, as a side effect, filter out chickens from enduring the subsequent wag_tail operation.

If we ignore exceptions from calls in the intra-procedural control flow graph, simply using exceptions to direct control flow is not enough to harm the result. The exceptions must be used to coordinate polymorphism by directing types along different paths. We argue it is reasonable to ignore these rare uses of exceptions in our control flow graph and that is what we do in our implementation (chapter 6). However, a conservative solution could assume all calls can throw an exception and an optimisation could investigate any call receivers that the flow analysis manages to resolve to see if exceptions are possible. In all cases it is safe to ignore interpreter exceptions that may occur non-deterministically because they cannot be used to coordinate polymorphism.

5.6 Approximating must-aliasing

In general, it is not possible to determine if two arbitrary expressions must-alias [21] but, depending on the semantics of the language, it may be possible to under-approximate it. Section 4.7.3 proves that limiting the relationship to instances of the same variable makes it possible to under-approximate must-aliasing if the analysis can prove that no execution step on any path between the two instances modifies to what the name is bound. This is still uncomputable, but the semantics of our toy language from section 4.10 allow us to under-approximate it statically for variables in the same method if there are no assignments to the variable on any intra-procedural control flow path between the two points.

The semantics of Python permit a similar analysis with certain caveats. Like the toy language, variables are named references to values (objects) and *local* variables cannot be modified outside their local scope.[3] Therefore, two occurrences of the same variable within a local scope without an intervening assignment in that scope are guaranteed to alias each other regardless of the code executed between them. Consider the following example:

```
def func(m):
    m = B()

x = A()
x.p()
func(x)
x.q()
```

The value that receives the call to p, x_5, and the value that receives the call to q, x_7, are guaranteed to be aliases regardless of what func and p actually do. The intervening code might modify the value to which x refers, but it cannot modify x to refer to something else. In this example, although func reassigns its parameter, it only affects the reference, m, which is local to the function's scope. When it returns to the calling scope, x still refers to the same value as before.

As with IntraNK from section 4.10, under-approximating must-aliasing

[3]Well, almost. There are ways to do it including aliasing and then modifying the locals() dictionary or using the **exec** statement, but the behaviour has changed between Python versions and is now all but forbidden.

between two variables in a procedure boils down to ensuring that there are no intervening assignments in the local scope. A number of techniques exist in the literature for this sort of analysis [3, 7].

We have to be a little careful. In Python, global variables complicate alias analysis, since intervening code is free to retarget the reference, as shown here:

```
def foo():
    global x
    x = B()

def bar():
    global x
    x = A()
    x.p()
    foo()
    x.q()
```

A variable must truly be local to be considered for must-aliasing by the intra-procedural kill analysis. Fortunately, unlike in some languages like JavaScript, which variables are local and which are global is a static decision.

As proved in chapter 4, being unable to establish aliasing in all cases does not affect the soundness of our results; underestimating aliasing only affects the precision of our inferred set of required features and therefore the precision of contraindication. Indeed, any sound computable alias analysis will necessarily compute an under-approximation [21]. More powerful analyses may be able to determine aliasing for a greater range of expressions, but this would require further investigation to see if in practice the cost of the analysis is worth the increased precision.

5.7 Class as upper bound

After propagating the required features, each variable has a static approximation of its required features, the 'added value' that enables contraindication of a flow-based type. But to use the feature for contraindication there must be some way to link it back to the classes being contraindicated.

For Python, this link is the **class** declaration, which lets the author define the methods that all instances will support when they are created.

```
class MyClass(MySuperclass):
    def a_method(self):
        print "I am a method of my class"
    def another_method(self):
        print "and another"
```

All instances of this example class are created with the two declared methods as well as any non-conflicting methods they inherit from the superclass hierarchy.

Flow-based type inference is sometimes called *concrete type inference* [2, 32] because it infers *concrete types*: sets of value creators, typically classes. Success types, on the other hand, are *interface types*: descriptions of a value's supported operations. In statically typed languages, these are compatible views of a type because classes completely define the operations supported by the values they produce and this cannot change at run time. Most duck-typed languages, however, support the run-time modification of classes. This does not affect our success types—required features are required regardless of whether we can map them back to a declaration—but using these features to contraindicate classes may cause us to do so erroneously if we are expecting to find all supported features in the class declaration.

Importantly, this problem does not reduce the contribution of our method for many applications of type inference, as it is also a problem for the existing flow-based type inference: Flow analyses may identify which class constructed an object, but this does not necessarily mean that the object has that class's declared feature at a given point in the program. For an application such as refactoring this may well cause the tool to break working code.

Like most existing work [28,41] and practical tools, we assume this particular aspect of dynamic behaviour does not occur. While this is not true in practice, it is a reasonable and necessary assumption to make when trying to provide any sort of tool that requires the static analysis of a dynamically typed program. Chapter 7 shows the effect that this assumption has on the quality of our results.

5.8 Closed world

In practice, no Python environment makes all program code available for analysis. For instance, the canonical Python interpreter, CPython, supports compiled modules implemented in C, as well as built-in classes and functions, none of which can be analysed. This has two main effects: First, we may miss some class definitions, causing the type judgements to include fewer types than can actually occur, thereby threatening completeness. Second, we may fail to see uses of a value that occur when passing it into a call to a compiled function, reducing the number of restrictive operations the inference analysis can use to narrow down the type, thereby affecting precision.

In our evaluation, we worked around the issue for built-in classes and functions by mocking them up in Python. Unlike their real-world counterparts, they are not functioning implementations, but simulate enough of the definition and operation of the built-in classes and functions that type definitions can be resolved from their interfaces, contributing to the type-narrowing process.

Compiled modules, however, are provided by third parties, so mock implementations are not a general solution. We do not attempt to provide mock versions of any compiled modules and this is reflected in the results given in chapter 7. A practical development tool could provide mock versions of the common third-party modules, such as GUI libraries, and potentially make this extensible so that third parties can add their own mock versions of their modules to the development environment.

5.9 Straying from the ideal

In this chapter we have discussed how the formal prerequisites from section 4.12 can be adapted to the semantics of Python, a practical language. Sometimes this means compromising on precision, but the compromise was for the sake of soundness:

- not considering fields as features and
- ignoring methods called indirectly.

Sometimes we are unable to find a compromise that maintains the soundness of the analysis, and we make a case for why these further compromises are justified in the context of actual program behaviour:

- reflection by calling a method and catching the resulting duck test failure (AttributeError),
- adding methods to a class other than through a method declared in the class definition,
- calling functions via method-call syntax,
- exceptions raised in a called procedure being used to coordinate polymorphism in their calling ancestors, and
- compiled modules hiding class declarations and data-flow paths.

Our results show that in practice actual behaviour either rarely uses the problematic features or does so in a way that leaves the types unaffected (chapter 7). While our formalism may demand the prerequisites of section 4.12, we argue that as long as violations are only minor, the technique can apply usefully to a range of less than ideal languages.

6 Implementation

In this chapter we present the implementation of type inference with contraindication that we developed for Python and used to produce the results in chapter 7.

We begin by describing our flow analysis implementation, which we base on DDP by Spoon and Shivers [41, 42]. Then we explain how we calculate domination and aliasing, the ways in which our implementation differs from our formalism and conclude with an outline of how we collect type definitions from the system to perform contraindication.

6.1 Design decisions

A major goal for our implementation is to use it to create a Python environment for the popular Eclipse IDE[1] and, although this has not yet been achieved, this goal influenced our design decisions.

Firstly, the implementation is written in Java rather than Python; Eclipse is written in Java and there is no easy way to combine the two languages.

Secondly, from the array of possible flow-based type inference techniques (section 2.5), we chose the demand-driven approach of Spoon and Shivers [41, 42]. No sound flow analysis for Python existed[2] and demand-driven type inference, being expressly designed for IDEs, seemed the obvious choice.

6.2 Flow analysis implementation

Spoon and Shivers developed their demand-driven technique, DDP, for the duck-typed object-oriented language Smalltalk, which in many ways is sim-

[1]http://www.eclipse.org

[2]Non-sound implementations appear in IDE tools such as PyDev (http://pydev.org/) and Rope (http://rope.sourceforge.net/).

ilar to Python. Nevertheless, building a flow analysis for Python is a time-consuming endeavour. Python is a large language and a sound flow analysis of real-world programs requires a complete abstract interpreter for Python, written from scratch.

The basis of the demand-driven analysis is that results are only calculated on request. A result is requested by posting a question, known as the root goal, to the analysis engine, which responds with a conservative approximation of the answer. But goals, including the root goal, may depend on other goals, which are also posted to the engine, so the answer to the root goal is only returned once all the sub-goals, upon which it transitively depends, have been answered. Furthermore, the dependency between goals can be circular, especially in a higher-order language like Smalltalk or Python, where data flow and control flow depend on each other, so each goal's answer is only provisional until the system reaches a fixed point.

Each goal starts with a trivially over-optimistic result and, as the analysis encounters evidence of this over-optimism, the goal's answer is refined to take that into account. When a goal's answer changes, the analysis engine places all the goals that depend on it onto a queue for re-processing so that they may refine their answers in light of the new information. The goals are carefully designed so that their answers only increase monotonically, so the analysis is certain to terminate. Once this happens, the goals are all "justified with respect to each other" [41] and the root goal's answer is known to be a conservative approximation of the true answer.

The algorithm of Spoon and Shivers [42] also supports goal pruning (the P in DDP), where it assigns a trivially pessimistic result to a goal if the goal meets particular criteria, for instance, taking too long to reach an answer. The effect of pruning a goal is that any subgoals it posted are now ignored and the analysis can continue as though that goal had a fully justified answer. This allows the analysis to bypass particularly hard questions based on prevailing run-time conditions without aborting the original request entirely.

Our implementation, however, does not use pruning, as we need a consistent level of precision between runs for the results in chapter 7 to be meaningful. Pruning compromises precision *dynamically*, so apparent variations in precision might have been caused by factors outside our control. Obviously, pruning is suited to interactive applications, so we are certain to add it when we integrate our implementation into an IDE.

Another difference is in the use of context. DDP is a context-sensitive analysis using the Cartesian Product Algorithm [2]. We chose not to make our analysis context sensitive both to keep the already complex flow implementation as simple as possible and to make the already slow analysis as fast as possible. In the future we would like to incorporate optional context sensitivity and compare the increased accuracy of contraindication with the increase, if any, from context sensitivity (section 8.3.1).

A final difference is that our flow analysis implementation does not do a temporary cast to resolve contradictions (section 3.2). Instead, it returns $\top$ and the cast becomes the basis of contraindication (section 3.6).

Although we include execution times in chapter 7 (table 7.2), we do not claim they are representative of the cost of a context-insensitive flow analysis. DDP is designed to favour performance for a single type query over performance for an exhaustive analysis [41]. We used DDP despite this because we intend to include our work in an IDE (section 6.1) and developing two separate flow analyses did not seem a sensible use of time.

6.3 Control flow graph

We approximate observed and required features on an intra-procedural control flow graph that we build from the abstract syntax of each function and method body. As described in section 5.5, to produce the results presented in chapter 7 our control flow graph ignores the possibility that calls might raise an exception rather than returning to the call site. This permits us to use an intra-procedural analysis without resorting to adding an exceptional transition from every call site—the conservative assumption. Nevertheless, our implementation is able to make the conservative assumption as well.

We calculate the intra-procedural control flow graph for each Python procedure (function, method, class and module[3]) in a single pass without first translating into three-address form. In the standard way [3] the graphs consist of basic blocks of statements which execute only sequentially and transition between blocks where execution may branch. Each control flow graph has an additional, empty entry block and two empty exit blocks: one

[3]In Python, classes and modules are also procedures. Both execute the first time a module is imported.

representing conventional termination or return and the other representing exceptions escaping the local scope.

Our implementation supports most Python syntax although generators are treated as functions which may not be correct, and both `with` statements and decorators are currently ignored. Recent language changes will require further study, but we intend to support contemporary Python in the near future.

6.4 Dominator analysis

We propagate the basic sets to other expressions to maximise the size of our frozen duck types. As described in section 4.7, observed features of a variable propagate to its *dominating aliases* and required features, to its *postdominating aliases*.

We calculate an under-approximation of dominating and postdominating aliases from the intra-procedural control flow graph using a standard domination algorithm to approximate true domination combined with SSA (section 6.5) to approximate must-aliasing. Chapter 4 describes at length why this combination of approximations leads to a sound approximation of observed and required features.

We adapted code from Android's Dalvik[4] bytecode compiler for our dominator analysis implementation.

6.5 SSA

We use static single assignment (SSA) [7] to perform alias analysis via kill analysis in our implementation, where our toy language used IntraNK. The choice was pragmatic; using SSA meant we could get a working implementation quickly. SSA is known to scale well, implemented in terms of dominators—an analysis we already have—and the implementation is textbook. We are certain that better analyses are possible; SSA is in no way fundamental to the approach.

SSA renames variables such that each variable has a unique definition. Therefore, any appearances of a local variable within a procedure sharing the same 'SSA name' are guaranteed to alias each other.

[4]http://code.google.com/p/dalvik/

As with much of the discussion of aliasing since section 4.7.3, the aliasing that SSA establishes is much stronger that the ideal, postdominating aliasing. Firstly, it is must-aliasing, which means, for example, that it will not give us the postdominating aliases inside a loop body if the alias is redefined in that loop (section 3.9). Secondly, the aliasing is only between variables of the same name, so unambiguous, unique assignments to different variables are not included. This reduces the precision of our method while keeping it sound. Nevertheless we believe this can be improved in the future (section 8.3.2).

6.6 Selectively inter-procedural propagation

The way we propagate features differs from that presented in Chapter 4 in one significant way: where the formalism is shown to be sound either for an inter-procedural or an intra-procedural control flow graph, we use a hybrid of the two, which we have not yet been able to formalise. We developed our intra-procedural formalism to avoid depending on a precise inter-procedural analysis. However, eventually we will use inter-procedural flow analysis to infer types which we then refine using our observed and required features through contraindication (sections 4.11 and 6.2). Although a tractable implementation of this analysis is not precise, it would be perverse not to take advantage of it when it is. Therefore, when the flow analysis is precise enough to infer a single receiver for a given call site, we can continue the analysis in the procedure body, mapping the arguments at the call site to parameters in the procedure. Required features that propagate up to the parameters can also propagate out of the call site back to the arguments and beyond. If, on the other hand, flow analysis is not able to uniquely resolve the receiver, we simply ignore it.

We argue that this does not affect the soundness of the approximation, as it is equivalent to inlining the procedure body and remains within the definitions of dominating and postdominating alias. Those definitions (definitions 13 and 14) require aliasing to occur between the (post)dominator and the (post)dominatee along any path that includes both points. They do not insist that the (post)dominator must *only* alias that program point, something that would prevent us looking into procedure bodies that might also be called from elsewhere with other arguments. Formalising this hybrid approximation and proving that it remains an under-approximation of postdominating aliasing is an open problem (section 8.3.2).

6.7 Contraindication

Our flow analysis, like most other flow-based type inference, infers types as a set of named classes. The observed features tell us which features the value definitely has and required features tell us which features the program cannot possibly survive without. If any of the classes in a variable's flow type lack one or more of the features, we know that, in any well-formed program, the variable cannot be an instance of that class, so we refine the type by removing it. This is the essence of contraindication.

When requested for a variable's contraindicated type, our implementation performs flow analysis to obtain the set of named classes. It processes each one by collecting all declared method names, resolving its superclass, and recursively collecting all method names defined in the inheritance hierarchy. Resolving the superclass may require further flow analysis because they are dynamic expressions in Python. We have never seen this in practice, however.

At this point the class and the success type have both been reduced to a set of method names at which point contraindication becomes a simple set operation; if the class signature is not a superset of the success type, it is contraindicated from the flow type.

Top A flow type that deserves special mention is $\top$, the most pessimistic type. Flow analysis, especially in a dynamically typed language, frequently encounters situations where it cannot reconcile conflicting information caused by imprecision and has no choice but to return $\top$ as the type (section 2.5.6). In theory, $\top$ is an infinite set which we cannot process a member at a time as we would normally. However, as an implementation detail of the flow analysis, we know all the Python modules that the program loaded directly or indirectly. Even if the flow analysis has lost track of which class flowed to a variable inferred as $\top$, we know it must come from a loaded module; classes cannot be instantiated without loading their parent module.

We harvest every class declaration in every loaded module once at the start of the analysis and use this to contraindicate $\top$ types. In fact, our results show that the majority of the types our analysis improved had previously been $\top$ (chapter 7). This goes some way towards showing how severe a problem $\top$ results are in flow analysis of dynamically typed languages.

Even if we had not had the list of loaded modules available to us, the analysis has all the source code available, so could have harvested all class declarations, albeit with a corresponding performance and precision penalty.

6.8 Computational cost

The flow-based concrete type inference (section 6.2) is an inter-procedural analysis and, as such, has at least cubic complexity in the size of the program [24]. The interface recovery analysis (sections 6.3 to 6.5) is calculated on an intra-procedural control-flow graph. The cost of constructing the graph is limited by the branching possible at non-call nodes, making it linear in the size of the program. Therefore, the cost of contraindication is dominated by the cost of the flow analysis.

However, it does not follow that contraindication adds negligible cost to an existing concrete type analysis, at least not the way we have implemented it. The reason for this is that the flow analysis implementation that we are using is demand-driven (sections 2.5.5 and 6.2) and we implemented our interface recovery analysis as a hybrid intra/inter-procedural analysis that, although largely intra-procedural, sometimes follows procedure calls if the inter-procedural flow analysis is able to resolve the callee uniquely (section 6.6). Because the flow analysis is demand-driven, attempting to resolve the callee may initiate *extra* flow analysis beyond that which was needed to infer the concrete type being contraindicated.

A compromise solution, which we might explore in the future, is to limit the interface analysis so that it only follows calls if the callee has already been resolved uniquely by the flow analysis, but does not initiate any new work.

7 Evaluation

We evaluate contraindication by measuring the effect it has on the results of a context-insensitive flow analysis in a range of real-world programs. The implementation we used for the flow analysis, the frozen duck type analysis and the final contraindication step are described in the previous chapter.

Our evaluation aims to answer the following questions:

1. Does flow-analysis with contraindication produce types that are more precise than flow analysis alone?
2. How large is the increase in precision?
3. Is there a benefit to using required features for contraindication over observed features?

7.1 Choosing the corpus

We chose eight programs to study, all drawn from open source Python projects.

ACE: a cosmogenic nuclide calculator;[1]

BitTorrent: a peer-to-peer file-sharing service;[2]

buzhug: a non-SQL database engine;[3]

Gadfly: a SQL database engine;[4]

Lyntin: a text-based multi-player game client;[5]

[1]http://ace.hwr.arizona.edu/
[2]Originally obtained as open source from http://www.bittorrent.com/. This project has recently been rendered proprietary.
[3]http://buzhug.sourceforge.net/
[4]http://gadfly.sourceforge.net/
[5]http://lyntin.sourceforge.net/

ReportLab Toolkit: a PDF document generator; [6]

Roundup: an issue-tracking system;[7] and

PySpaceWar: an interactive, single-player game.[8]

This diverse population varies in both size and application domain. We found the projects through SourceForge,[9] the Python Wiki[10] and the Python Package Index.[11] We selected them based on several criteria:

Size: We selected the programs to cover a range of sizes (table 7.2) in case program size affected contraindication. However, we only considered projects with fewer than 100,000 lines of code to allow the flow analysis to complete in a reasonable time. Nevertheless, even with this limit, we were unable to analyse every expression in a reasonable amount of time, so we restricted the analysis to consider only call receiver types (section 7.1.1).

Working order: Our analyses require that all code be present, so we needed to be reasonably sure the programs were in working order. By this we mean that all dependencies, both on third-party libraries and internally, were satisfied. This is different from requiring the program to be well formed, something we assume by virtue of these being publicly available open-source programs.

We excluded any program whose internal dependencies could not be satisfied without an installation phase, as well as any programs whose third-party dependencies were not easy to install.

Pure Python: We excluded any program that included compiled modules, as this means the analysis does not have all the source code available for study. This restriction only extended to compiled modules included with the program since, in practice, many projects depend on third-party modules (including those in the system library) that themselves depend on compiled components.

[6]http://www.reportlab.com/software/opensource/rl-toolkit/
[7]http://roundup.sourceforge.net/
[8]http://mg.pov.lt/pyspacewar/
[9]http://sourceforge.net/
[10]http://wiki.python.org/moin/MostPopularPythonProjects
[11]http://pypi.python.org/

Table 7.1: Effect of contraindication on the precision of types inferred through flow analysis.

PROGRAM	FLOW$_\top$	OBSERVED			REQUIRED			COMBINATION		
		IM	IM$_\top$	$\Delta_{\neg\top}$	IM	IM$_\top$	$\Delta_{\neg\top}$	IM	IM$_\top$	$\Delta_{\neg\top}$
ACE	45.3%	8.0%	77.0%	3.3%	44.9%	91.2%	7.9%	51.3%	88.2%	11.0%
BitTorrent	49.1%	8.8%	83.4%	2.8%	54.2%	90.6%	9.8%	54.6%	90.0%	10.5%
buzhug	42.8%	3.6%	88.9%	0.7%	44.5%	95.9%	2.9%	44.9%	95.1%	3.6%
Gadfly	50.5%	15.6%	65.1%	11.5%	59.0%	85.6%	17.9%	59.1%	85.4%	18.1%
Lyntin	44.6%	6.0%	92.0%	0.7%	47.8%	93.3%	4.8%	47.8%	93.3%	4.8%
PySpaceWar	44.2%	5.9%	93.0%	0.7%	45.7%	96.7%	2.7%	45.8%	96.4%	3.0%
ReportLab	44.6%	12.7%	92.0%	2.1%	53.9%	95.7%	4.7%	54.0%	95.6%	4.8%
Roundup	48.9%	19.1%	55.2%	19.3%	64.8%	75.5%	35.7%	65.0%	75.2%	36.2%
	47.1%	10.0%	80.8%	5.1%	52.4%	90.6%	10.8%	52.8%	89.9%	11.5%

7.1.1 Inference targets

We could not infer a type for every expression in the corpus exhaustively; the flow analysis just does not scale. So we picked a particular kind of expression for which we infer types: method call receivers, i.e., the expression on the left hand of the full stop in a called attribute. For example, receiver in this case:

```
receiver.method()
```

Method call receivers are particularly interesting for program analysis but notoriously hard to establish.

Limiting the expressions this way brought the number of types inferred to between 497 and 9448 per project, which we could infer in a reasonable amount of time (see table 7.2).

7.2 Result of contraindication

Table 7.1 summarises the results of our analyses for each of the eight case studies. FLOW$_\top$ is the percentage of call sites that pure flow analysis inferred as $\top$, meaning it was unable in those cases to provide any sort of precise type. We explain why this arises and why it is so common in section 2.5.6.

The next three sets of columns show to what degree our contraindication using frozen duck types is able to improve upon pure flow analysis. The three sets show the relative effect of contraindication when the frozen duck types are made using only observed features (OBSERVED), using only required features (REQUIRED) and using the union of both (COMBINATION). Each set is divided into three columns.

Column IM gives the percentage of all types inferred by flow analysis that contraindication was able to refine. This does not take into account the magnitude of the improvement, just its frequency.

Column $\text{IM}_\top$ gives the percentage of those refined types that had been originally inferred as $\top$ by flow analysis. In other words, this column indicates to what extent the least useful result from flow analysis is improved. These correspond to an improvement of infinite magnitude.

Finally, the column labelled $\Delta_{\neg\top}$ gives the average magnitude of the improvement in the size of the types inferred by flow analysis when that improvement is not infinite. The magnitude of the improvement is the percentage decrease in the number of abstract values in the non-$\top$ concrete type. In other words, this column indicates by how much, on average, contraindication is able to further refine those types for which flow analysis was able to provide some amount of precision.

Looking at the last row of table 7.1, which provides the results averaged across the eight case studies, we can see that all three applications of contraindication are significantly better able to provide types that are more precise than flow analysis alone.

Contraindication based on required features alone led to an average of 52.4% of the types inferred by flow analysis being made more precise. The vast majority of those improved, 90.6%, had been inferred as $\top$ by flow analysis. The remaining types (i.e., the non-$\top$ types) were reduced in size by an average of 10.8%.

In contrast, when contraindication is based on observed features, only 10.0% of the flow analysis types improved. As with required features, the vast majority of these, 80.8%, had previously been inferred as $\top$. Of the remainder, the size of the inferred type was only reduced by 5.1%.

When we based contraindication on the combination of the observed and required features—the full frozen duck type—the results were only slightly better than using required features alone: on average 52.8% of the flow analysis types improved and, of those that had not previously been $\top$, they

Table 7.2: Relative run times of contraindication using only observed features, only required features and both kinds of features together. All tests were carried out on an Intel Core 2 Duo 3GHz system with 4GB RAM and a 3GB Java heap size limit.

Program	LOC	Call sites	Run time (seconds)		
			Observed	Required	Combination
ACE	18039	5509	3219	2558	2500
BitTorrent	69969	8888	4416	4754	4478
buzhug	1970	497	298	288	279
Gadfly	14704	2078	301	310	273
Lyntin	12306	2486	433	405	416
PySpaceWar	4392	731	253	267	258
ReportLab	49728	6091	800	854	808
Roundup	39369	9448	929	888	1080

improved by 11.5%.

7.3 Analysis

The improvement made by the combined analysis is not the sum of that made using the observed and required features separately, and we did not expect it to be. The two sets of features can have members in common and these members can only contribute to contraindication once in the combined analysis.

We also expected the result using only required features to show a better improvement than that using only observed features. Our target expressions are all method receivers, so every target was bound to have at least one required feature: the method being called directly on the expression being typed. This method is not an observed feature of the expression because the call happens after reaching the expression.

The difference between the flavours of contraindication is significant: 10.0% for observed features compared to 52.4% for required features. This suggests that much of the improvement comes from the single method directly on the expression. Non-variable expressions do not have observed features because our analysis cannot establish the aliasing requirement needed to create the basic set (section 4.7.2). We suspect that if we were to repeat the analysis with the targets limited to variables, the analyses would produce more similar results.

We were not expecting the combined analysis to vary so slightly from the result using only the required features. One explanation might be that when two or more methods are called on the same variable—the only situation that leads to observed features in this study—the same method tends to be called each time. Whether this is actually the case is itself an interesting question (section 7.4).

Every program studied showed a significant improvement in the precision of inferred types. Most of this improvement came from refinements of types on which flow analysis had failed entirely. The large percentage of refined $\top$ types, combined with the small percentage by which contraindication reduces the non-$\top$ types, seems to imply that when flow analysis infers a non-$\top$ type it does so fairly precisely. However, the true benefit of contraindication for non-$\top$ types may be obscured by the sheer number of them inferred by flow analysis. We suspect that if the flow analysis were improved by, say, context sensitivity, then the benefits of contraindication for refining non-$\top$ types would become more apparent.

The results we obtained and describe here are from an implementation of type inference applied in a single pass. We suspect that we can obtain an even better result by interleaving passes of flow analysis and contraindication, iterating until reaching a fixed point, with each pass providing more precise types to improve the next. This is an important idea to explore in future work (section 8.3.3).

7.4 Threats to validity

The evaluation we present here is vulnerable to a number of threats to its validity. The first has to do with the choice of case-study subjects. We made an effort to find a broad set of programs, but that set is indeed small and may be skewed by the fact that they are (or were) developed as open source projects. We cannot, therefore, assert that our results extend beyond the set of chosen subjects, but the universally positive results from them increases our confidence in the method.

A second threat arises from the fact that we were unable to use our implementation of flow analysis to infer types for every expression in a

Table 7.3: Verification of the results by manual inspection of a sample from the COMBINATION analysis.

PROGRAM	SAMPLE SIZE	ERROR	COMPILED MODULE	RT METHOD ADDITION	EVAL	RT CODE GENERATION
BitTorrent	57	8.8%	40.0%	60.0%	0.0%	0.0%
PySpaceWar	30	16.7%	80.0%	20.0%	0.0%	0.0%
buzhug	22	4.5%	100.0%	0.0%	0.0%	0.0%
ACE	72	12.5%	0.0%	66.7%	22.2%	11.1%
		10.6%				

reasonable amount of computational time. We therefore limited the scope of our evaluation, focusing on method call receivers. The threat, then, is that the results of the evaluation are not representative of arbitrary expressions. In particular, contraindication on method call receivers will always have at least one required feature to take into account: the method called on the expression being typed. The same is not true of observed features, which can only consider method calls on strictly dominating aliases. While it is clear that using required features produces many more refined types for method call receivers than using observed features alone, we cannot yet say how large the improvement would be for arbitrary expressions.

A final threat to validity is that our method or its implementation may be flawed. To counter this threat we verified our results by manually inspecting a random sampling of the call sites from the version of the analysis using both observed and required features—the version most likely to over-contraindicate the type. We report the results of this exercise in table 7.3. For each case study subject we give the number of samples and the percentage of those samples found to be in error. An error is defined to be a case where the inferred type was too narrow and, therefore, incomplete. We identified four sources of errors in our results: (1) a class definition hidden within a compiled module; (2) a method being added to a class other than by declaration; (3) the use of a Python `eval()`, which hides where a constructor is called; and (4) a run-time declaration of a class. The first, second, and fourth of these are associated with the restrictions discussed in section 5.9. The third is a weakness of flow analysis.

Looking at the results, we can see that the main cause of errors varied by program, but that the majority were caused by compiled (GUI related) modules obscuring class definitions, which renders contraindication blind to their contribution to the analysis. Those problems can be resolved by increasing the set of mock interfaces, as discussed in section 5.8. The other major cause of errors can be attributed to methods being added to classes at run time. In fact, all the errors of this type for ACE were caused by a single method added to a class via attribute assignment rather than via method declaration. In this particular case it would be easy to work around the problem, since the assignment occurs in the class definition and simply aliases a declared method. However, in general the problem remains due to our assumed restriction.

On average, 10.6% of the results produced by contraindication were in error, so the IM results for the COMBINATION in table 7.1 may potentially be that much lower. But even taking this into account, the improvement in precision from contraindication is still quite significant.

7.5 Application: code completion

Several areas of programmer assistance that typically use conventional flow-based type inference would gain from applying our approach. We discuss the benefits of one such application here. Other applications are briefly described in section 8.4.

Programming environments assist the user in their task by suggesting possible *completions* at appropriate moments. These completions are snippets of code which the user chooses from the list of suggestions. The assistance is two-fold; firstly, the most likely choice should be easier to select than to type manually; secondly, the suggestions may reveal possibilities the user was not aware were available. Our approach can improve the quality of this type of assistance.

The typical scenario we have in mind is where the user types code such as:

```
receiver.
```

Upon typing '.', the system presents the user with a list of suggested names with which they could complete the statement.

Assuming we want the list to include only correct suggestions—the current state of type inference in dynamically-typed languages is such that environments have often resorted to unsound heuristics—the programming environment must infer the type of receiver to calculate completions that are valid for all values that may appear there. The difficulty is that imprecise inference may obscure valid suggestions. For instance, in the example in section 1.3 on page 11 let us suppose the user has inserted a new line between lines 16 and 17 started typing:

```
d = p.
```

At this point, the environment should be able to suggest foo() and bar() as completions, but an analysis based purely on flow-based type inference would not be aware of this due to the appearance of A and B in the inferred type of p.

Our analysis improves the results by using recovered interface information to refine the type and reveal the previously obscured completions. In this case it uses the later calls to foo and bar, on lines 17 and 18, to refine the type to just {C}, which would allow an environment to use the methods of class C as the basis of its list of completions: foo() and bar().

Of course, to get any benefit from our approach for code completion, the code must already make use of distinguishing features of the values for which completions are needed. At first glance it may seem that the only additional completions that can be suggested are for features that the programmer has already used.

Even if this were the case, revealing these suggestions is still a significant improvement. The features that have already been used may not have been used recently or may be used far away from the location being edited, and, as code is not written top-to-bottom in practice, features used after the point being edited also result in improved suggestions, as in our example. Code making up a program is also usually not written by the same person. Revealing features used elsewhere by another author is very useful, especially when this usage is hidden inside standard library code.

However, it is not the case that suggestions are limited to features already used. Use of one feature reveals other features that are coupled to it. For example, using a feature push may reveal a feature pop because use of one feature is likely to limit the type to classes with the closely-coupled method. If the feature used is unique to a class, it allows the environment to use the

Table 7.4: Comparison of code completion using different inference methods. The data show the percentage of completions that would have been successful using each method.

Program	Flow	Interface	Contra
ACE	16.0%	61.4%	86.2%
BitTorrent	28.6%	47.7%	77.9%
buzhug	30.0%	52.8%	79.5%
Gadfly	25.3%	38.8%	68.0%
Lyntin	23.9%	42.8%	66.6%
PySpaceWar	34.5%	48.3%	79.3%
ReportLab	74.0%	63.0%	92.1%
Roundup	24.3%	51.6%	71.2%

entire interface of the class to suggest completions.

A limitation of our approach when applied to code completion is that it assumes a well-formed program. If a user is in the process of creating the code, they may attempt to use a feature of a value that does not exist. This would mislead our analysis and cause it to narrow the type too much and present suggestions that are not valid. A solution to this problem might be to base feature recovery only on the features in use at some sort of checkpoint such as the last time the code was saved or the last successful run of the unit tests.

7.5.1 Experimental analysis

To get an idea of contraindication's practical impact on code completion, we compare code completion based on three different sources of information: flow analysis alone, recovered interfaces alone and the contraindicated types. For every method call site in our test corpus, we simulate code completion as though the user had not yet typed the name of the method. Completion is considered successful if the suggestions that would have been made includes the method that was actually chosen.

The results of our analysis are in table 7.4. The data are the proportion of completions that would have been successful. In column Flow, suggestions were based on flow analysis alone. In column Interface, they were based on the interface recovered using the COMBINATION analysis (see section 7.2), without reference to flow information. In column CONTRA, they are based on the flow types contraindicated using the recovered interface.

The results show a clear benefit to using recovered interfaces to contraindicate types for code completion. It is also interesting to note that, in all programs except ReportLab, code completion based on the recovered interface alone includes the chosen method more often than when based on the results of flow analysis.

7.5.2 Costs associated with code completion

Although the cost of our analysis is no different for this application than for others, the nature of code completion is that it occurs frequently, and each time a suggestion is chosen by the user the code is mutated. Ideally, we do not want to perform the entire analysis from scratch every time this happens.

Code mutation can invalidate the results of a previous run of our analysis in two ways. Mutation can introduce new data-flow paths that let additional classes of value reach a variable. In this case, the concrete type produced by the earlier flow analyses would no longer conservatively overestimate the possibilities in the modified program. Mutation can also remove features from a class, the result being that, although the previously inferred types remains an over-estimate, suggestions made on the basis of previous analysis of the class's interface may be invalid as the interface has changed.

The second problem is easily resolved by re-running just the analysis that maintains the database of class interfaces—a cheap analysis—on the classes whose features were removed. Any future suggestions will then be valid. new flow analysis is not necessary because data-flow has not changed. The results of the previous interface recovery analysis are still valid because the uses of features remains unchanged. The contraindication step, also, does not need to be re-run because it filters classes out of the concrete type based on features that are used but not present in the declaration. Removing a feature after the analysis runs means it may miss an opportunity to refine the concrete type, but the type will still be a conservative approximation. Nevertheless, as contraindication is a simple set operation and, therefore, fast, it makes sense to rerun it using the updated results of the class interface analysis.

A limited solution to the first problem may lie in the nature of code completion suggestions, which typically insert code that uses a feature without affecting *local* data-flow.

```
def func():
    x = something()
    x.g()
    x.h()    # inserted by code completion
    if something():
        x.i()
```

In this example, the insertion of x.h() by code completion cannot change the concrete type inferred for x anywhere in the function. x is a local variable, so any new data-flow resulting from calling method h cannot assign additional classes of value to it. Even if calling x.h() modifies global state, causing subsequent calls to something() to return additional classes of value, the previous flow analysis that producing the return type for something() must already have included those classes in the concrete type of x.

This means that the results of the flow analysis are still valid. The results of the interface recovery analyses are also still valid because the recovered interface of x will still have all the recovered features, though it may not be as precise as it could be, because it does not include the new feature use, h(). As both these analyses are still valid, the previous results of contraindication are still valid as well. In general, however, changes to data-flow will require all analyses to be re-run if invalid suggestions are to be avoided.

Code can be mutated in other ways that do not invalidate the results. For example, adding a feature to a class means that previous contraindication may have refined a concrete type to exclude the class because it previously lack the feature. This does not invalidate those previous results unless data-flow is also modified to flow values of the new class to the point where the type was refined. If the class was previously correctly excluded at a point, adding a feature to the class will not make that judgement incorrect. This hints at the interesting possibility that the analysis could be made to produce more precise types if allowed to ‘watch’ the user programming than it could by analysing the code from a cold start.

8 Conclusion

For applications where type correctness is known or can reasonably be assumed, contraindication works well to infer useful static types. The types are useful in the sense that they can substantially improve the quality of certain programmer aids, such as refactoring or code completion for programs written in dynamically typed languages. We have taken a critical step toward demonstrating this by developing the theoretical underpinnings of contraindication, showing how the method can be realised in Python and evaluating how well it improves upon existing techniques.

8.1 Open problems

There are a number of problems we do not yet know how to solve. Among them are the issues arising from the difference between the ideal language semantics we assumed when we formalised our approach (section 4.12) and the actual semantics of practical duck typed languages such as Python (chapter 5). We summarise the compromises in section 5.9.

In particular, languages where exceptions can be thrown at any time pose a problem for our approximation of required features that relies on postdomination. And even if such non-deterministic exceptions are ignored, it seems it would require an inter-procedural analysis to remain sound in the face of exceptions used to coordinate polymorphism. In section 5.5 we argue why it is reasonable to ignore all such exceptions and, indeed, we did not find any errors caused by doing so when we sampled our results (section 7.4). Nevertheless, it remains a problem with the approach.

8.2 New avenues

Required features are defined in terms of TROUBLE, the behaviour that a well-formed program is guaranteed to avoid. In much of this thesis we have taken TROUBLE to mean a run-time type error and, therefore, that well-formed programs are type-correct programs. But nothing about TROUBLE means that it has to correspond to an error state. In section 5.2 we saw that it does not even have to be a halting state. All that is necessary is that it must be reasonable to promise that the behaviour never happens and the syntax makes it possible to predict when it might occur.

Take squeaking as a ridiculous example: imagine that the language includes a syntactic form that guarantees a program will squeak if certain run-time conditions hold at that point. Letting TROUBLE be all squeaking configurations and well formed be a promise that the program never squeaks, we can guarantee that the run-time conditions leading to squeaking never hold. This is a more general form of the ideas behind required features, where the syntactic form are attribute requests on a variable and the run-time conditions are that the attribute is missing from the dereferenced variable.

This obviously requires more thought. Squeaking is not a particularly practical example, but it illustrates a point that there may be applications, perhaps in relation to testing, where one knows more about the program than is evident from the source code, and being certain of the absense of some behaviour can allow you to direct program analysis more precisely.

8.3 Future research

8.3.1 Improved metrics

Verification We verified our results by sampling them and manually inspecting the source code to ensure that only those types included in the result could flow to the expression (section 7.4). Being dynamically typed makes this labour intensive because values, including classes, can flow unrestricted through programs, requiring a complete understanding of the source. The result is that we were only able to sample 1% of our results for half our programs.

Run-time inspection might offer another way to verify the inferred types. This kind of analysis monitors a running program and records the exact types of object seen at an expression. Such analyses even form the basis of type inference engines in their own right (section 2.2). However, these analyses are only as good as the test suite or user-interaction excercising the program and, therefore, should be used in addition to manual inspection. Nevertheless, they would add to the level of confidence in the results.

Context sensitivity Once we add context sensitivity to our flow analysis implementation, it would be interesting to compare the relative benefits of context sensitivity and contraindication, both in terms of pure precision as well as the performance/precision trade off. And, assuming we find context sensitivity is not hopelessly slow, it would be interesting to see if contraindication still has an appreciable increase in precision when applied on top of a context-sensitive analysis. We suspect it will because even a context-sensitive analysis will not be able to recover types for items in heterogenous containers (section 2.5.6).

8.3.2 Extending the formalism

We implemented contraindication such that, if flow analysis is able to resolve a function call to a single receiving function, the arguments passed to the call inherit the required features of the parameters to which they are passed (section 6.6). We have argued that this is correct as it is equivalent to inlining the call. However we have not yet shown this to be the case in our formalism (chapter 6).

Extending this, when flow analysis infers multiple receivers for a call, we would like to let arguments inherit the intersection of the required features of the parameter to which they are passed. Again, we believe this to be correct, but have yet to prove it formally.

Finally, in the most general case, we would like to improve our approach so that we reason about which *features*, rather than program points, dominate or postdominate an expression. This would improve the precision of the analysis by capturing, for instance, the example at the end of section 3.7, where a feature appears on both branches of the conditional but is ignored by our current analyses because the program points it appears at do not, individually, dominate.

8.3.3 Iteration

The implementation we describe in chapter 6 and use to produce the results in chapter 7 performs flow analysis to render a type for an expression followed by frozen duck type analysis and, finally, uses the latter to refine the former through contraindication. This already results in a significant improvement in precision.

However, we believe the true potential of the approach lies in feeding the results of the contraindication back into the flow analysis and iterating the analyses until the type reaches a fixed point. This will require careful thought to ensure the flow analysis terminates.

8.3.4 Richer features

Currently, our frozen duck types only consist of method names from the observed calls. In the future we could add information about the number of arguments or the keywords used. We have not done so yet because the semantics of parameter passing in Python are not straightforward. We already discussed the difficulty of including named fields. We could include features such as callability and iterability based on observations of those operations as well. In general, an analysis can use any property whose absence will cause some failure as a feature.

8.4 Practical applications

We explain in section 6.1 that our aim is to incorporate our implementations of flow analysis based on DDP [41, 42], frozen duck type analysis and contraindication into the Eclipse IDE. IDEs are essentially the amalgamation of many separate tools, many of which could benefit from our analyses including:

API documentation: Even in the absence of flow analysis, frozen duck type analysis can be performed inter-procedurally to produce an under-approximation of the set of features that either will be or must be satisfied at an expression. This is useful when documenting parameter and return types at an API. When performed on a parameter, the analysis equates to required feature analysis (section 3.8) and documents features that the caller must ensure the argument they pass provide. When performed on a return value, the analysis is observed feature analysis (section 3.7) and documents features the return value is guaranteed to support.

This is particularly useful in a duck typed language as type compatibility is decided on the basis of features rather than named types. Nevertheless, frozen duck types are similar to success types [23] and *under*-approximate the set of supported features, so do not guarantee that an argument will be compatible even if all the features are present, nor that the returned value will be incompatible with some operation requiring more features than the frozen duck type promises.

Code completion: Firstly, frozen duck type analysis can, again, operate in the absence of flow analysis to provide code completion suggestion to the user as they program. In Python this would happen when the user types a full stop after a variable name:

```
var.
```

At this point the tool assists the user with a list of features supported by the variable. The list might appear immediately as a drop down list or on demand as a key stroke that cycles the possibilities.

Of course, this is best done by a combination of frozen duck types and flow analysis through contraindication. That way the assistance benefits from the extra methods found through the class definition without losing the ability to assist when flow analysis inferred $\top$, which it does almost half the time (section 2.5.6).

Refactoring engine: Arguably the most pressing need for better flow analysis in dynamically type languages comes from *refactoring*, the process of making behaviour-preserving changes to the code wholly or partially in the absence of direction from the user. Such tools depend on an analysis that is both sound and precise:

- Sound, because the refactoring engines guarantee that their code transformations do not alter program behaviour. Unsound analyses mislead the refactoring engine, such that this guarantee is falsely made.

- Precise, because the refactoring engine uses the flow analysis as an impact analysis. Its purpose is to determine the smallest set of changes that will effect the desired transformation without changing behaviour. This is important because not only does the precision determine the scale of the changes, it often determines whether it is possible at all because much of the code in any system—the libraries for instance–may not be changed. If that were not enough, an imprecise analysis forces the refactoring engine to reason about non-sensical changes that arise as a result of the contradictions caused by imprecision (sections 1.4 and 3.1). These are precisely the kinds of contradictions we developed contraindication to mitigate.

Type checker: Although explicitly not one of our aims, parts of the analyses do lend themselves to type checking. While no analysis can result in a type *safety* analysis for a dynamically typed language that guarantees the absence of run-time type errors, flow-based type inference can be used to find definite errors in such languages when, for instance, none of the classes in the inferred type supports the operations being performed.

We have already likened required features to success typings, which were developed to find "type clashes" [23] in Erlang programs. Although we use required features for a different purpose, they underapproximate the set of feature requests to which a program will subject a value, so can be used to detect when a value will be incompatible with those requests.

Both these uses are subject to an assumption that classes and values have fixed sets of supported features. When that assumption does not hold, either analysis may produce false positives. A further analysis, perhaps similar to Anderson et al. [6] may mitigate this.

Lint: Unlike type checkers, bug finders may produce false positives because their output takes the form of warnings and advice to the user, rather than definite errors. The *lint* family of analysis tools fits into this category, for example Pylint[1] for Python.

[1]http://bitbucket.org/logilab/pylint

> Our flow analysis implementation would be a good basis for such tools. The analysis can be overly strict and behave like a static type safety analysis by flagging a potential run-time type error if *any* member of the set of classes in the inferred type fails to support the operations being performed.

In the absence of certain pathologically dynamic language features, such as `eval` and run-time code generation, which it cannot reason about, our implementataion of flow analysis, based on DDP by Shivers and Spoon, produces sound types unlike other IDE tools we have come across for Python (e.g., PyDev[2] and Rope[3]). This will make its assistance more reliable and we hope it will engender greater programmer confidence, as the assistance will no longer mislead. This is particularly critical for refactoring because incorrectly inferred type information causes refactoring to break the code as it changes it.

On the other hand, our current implementation is likely to be slower and less precise than the unsound inference engines. The challenge will be to achieve a level of precision that users are willing to accept in return for correct results. We believe our contraindication approach is the key here (sections 3.6 and 6.7).

Although contraindication is not sound for Python, as the language semantics do not match the prerequisites (section 4.12 and chapter 5), the problem is limited to certain uncommon behaviours (section 5.9). Nevertheless, we are likely to integrate contraindication as an optional feature, enabled by default for descriptive applications like API documentation while disabled by default for generative applications like refactoring.

8.4.1 Remaining work

Much of the work towards a useful Python development tool is already complete, but before we can produce a practical environment some more will have to be done.

Compiled module simulation Our results in table 7.3 show compiled modules are a significant source of errors in applying our analysis to Python.

[2]http://pydev.org/
[3]http://rope.sourceforge.net/

Compiled modules and built-in code violate one of the main assumptions both flow analysis and contraindication rely upon: that all source code is available for scrutiny. The solution is to simulate their behaviour using mocked implementations, special-case treatment in the analysis engines or annotation.

We simulated the built-in language functions using a combination of the first two. In any practical tool, we would need to simulate the compiled libraries as well and this may best be done through mocking and annotation because that permits third-party library developers to create their own descriptions of their library's behaviour that our tool can consume.

Context sensitivity Our implementation of flow analysis differs from DDP [42] by omitting context sensitivity from the analysis (section 6.2). We did so to produce a simple, fast implementation. However, the context-insensitive analysis infers $\top$ almost half the time. Although contraindication improves this, we would like to reintroduce context sensitivity to the implementation and measure the impact. In particularly, we would like to study whether any improvements due to context-sensitivity are still evident when combined with a pruning criterion that bounds the permitted execution time.

Pruning Pruning is the second part of DDP that we omitted from our implementation. We did so to produce deterministic results. However, an interactive environment is not suitable for analyses with unbounded run times. Pruning solves the problem by arbitrarily terminating attempts to obtain a more precise result if a timeout is reached (section 6.2).

Carefully compromised model In this thesis we have focussed on developing an approach to type inference that is provably sound for an ideal language (chapter 4). Even when applying our work to a language that was not ideal, we attempted to keep the result sound, sacrificing precision and performance wherever necessary in order to do so and only compromising soundness when the language semantics left us no alternative (chapter 5). This, in part, was motivated by our experiences with existing development environments for dynamically typed languages, which erred, instead, on the size of maximum assistance but as a result do more harm than good.

That said, when applying this analysis to a language of the size and flexibility of Python, there is an argument to be made that relaxing the soundness requirement in a few carefully chosen places to improve performance or precision is necessary for a practical tool. The important point is that these should be carefully chosen such that they lead to a large gain in performance or precision while compromising only rarely used corners of the language.

We are keen to explore one such compromise in the way we model class inheritence in Python. Contraindication needs to aquire the names of the declared methods of every loaded class, including any methods defined in any superclasses. In Python, a class's superclasses are arbitrary expressions resolved at runtime as the module containing the class is imported. Although we have never seen anything but named classes appear in the list, this still poses a problem because that name is a variable like any other and deciding what it contains, in general, requires flow analysis. Our implementation of contraindication invokes flow analysis to do precisely that and preliminary results (not included in this thesis) suggest that almost all the extra flow analysis cost incurred by the contraindication phase of our analysis is caused by superclass resolution.

We suspect the reason flow analysis is so expensive is that class names are almost always global variables in Python and global variables can be modified from any other module that imports the module in question. Our flow analysis models this possibility faithfully by inspecting all those other modules and analysing anywhere to which that global variable could flow in order to find all its definitions. Needless to say, this is expensive and the situation it models is unlikely: the module containing the class would have to import another module that, possibly transitively, reached back into the partially imported module and bind a value to the global variable from which the class inherits. We suspect that ignoring the possibility that these global variables can be modified from outside the module will lead to a significant speed improvement for contraindication with no effect on the correctness of the result.

8.5 Languages

Our approach is sound for duck typed languages that satisfy our ideal prerequisites (section 4.12). But, as far as we are aware, there are no practical languages that actually do so. Nevertheless, most of these prerequisites are commonly assumed in the literature. In particular, the assumptions that classes and features have fixed sets of features is critical not only for our own analysis but also for the other applications of flow analysis described in section 8.4.

The results of our manual inspection (table 7.3 on page 135) show that, despite the compromises needed to use the approach with Python, the refined types produced by our implementation are largely correct. Our comparison of type inference both with and without contraindication (table 7.1 on page 131) show that contraindication leads to a significant increase in precision, particularly when performed using required features as the basis of the frozen duck types. There are several other duck-typed languages with similar semantics to Python, notably Ruby and Smalltalk, and we expect they would benefit similarly.

Bibliography

[1] Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. Flow-sensitive type recovery in linear-log time. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, OOPSLA '11, pages 483–498, New York, NY, USA, 2011. ACM.

[2] Ole Agesen. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26, London, UK, 1995. Springer-Verlag.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[4] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: A step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[6] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 428–452. Springer, 2005.

[7] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, 2nd edition.* Cambridge University Press, 2002.

[8] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, July 1998.

[9] John Aycock. Aggressive type inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[10] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[11] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM.

[12] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[13] 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*, POPL '98, pages 222–236, New York, NY, USA, 1998. ACM.

[14] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[15] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[16] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[17] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the*

20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software, ESOP'11/ETAPS'11, pages 256–275, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. Type harvesting: a practical approach to obtaining typing information in dynamic programming languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1282–1289, New York, NY, USA, 2011. ACM.

[19] Alex Holkner and James Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, pages 19–28, Darlinghurst, Australia, 2009. Australian Computer Society, Inc.

[20] Suresh Jagannathan, Stephen Weeks, and Andrew K. Wright. Type-directed flow analysis for typed intermediate languages. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, pages 232–249, London, UK, UK, 1997. Springer-Verlag.

[21] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[22] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams.* PhD thesis, McGill University, January 2006.

[23] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 167–178, New York, NY, USA, 2006. ACM.

[24] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 305–315, New York, NY, USA, 2010. ACM.

[25] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[26] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '92, pages 329–349, London, UK, 1992. Springer-Verlag.

[27] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, July 1995.

[28] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.

[29] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems.* John Wiley and Sons Ltd., Chichester, UK, 1994.

[30] David J. Pearce and James Noble. Structural and flow-sensitive types for Whiley. Technical report, Victoria University of Wellington, 2011.

[31] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[32] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '94, pages 324–340, New York, NY, USA, 1994. ACM.

[33] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[34] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

[35] Michael Salib. Faster than C: Static type inference with Starkiller. In *PyCon Proceedings*, pages 2–26. SpringerVerlag, 2004.

[36] Olin Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174, 1988.

[37] Olin Shivers. Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned. *SIGPLAN Notices*, 39:257–269, April 2004.

[38] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2007.

[39] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[40] Dorai Sitaram. Teach yourself Scheme in fixnum days. Website.

[41] S. Alexander Spoon. *Demand-Driven Type Inference with Subgoal Pruning.* PhD thesis, Georgia Institute of Technology, December 2005.

[42] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In Martin Odersky, editor, *Proceedings of the* 18th *European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 51–74, Oslo, Norway, June 2004. Springer.

[43] S. Alexander Spoon and Olin Shivers. Dynamic data polyvariance using source-tagged classes. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 35–48, New York, NY, USA, 2005. ACM.

[44] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.*, 19(1):48–86, January 1997.

[45] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[46] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

[47] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

[48] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 99–117, London, UK, 2001. Springer-Verlag.

[49] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 377–388, New York, NY, USA, 2010. ACM.