The Ins and Outs of Gradual Type Inference

Aseem Rastogi
Stony Brook University
arastogi@cs.stonybrook.edu

Avik Chaudhuri Basil Hosmer
Advanced Technology Labs, Adobe Systems
{achaudhu,bhosmer}@adobe.com

Abstract

Gradual typing lets programmers evolve their dynamically typed programs by gradually adding explicit type annotations, which confer benefits like improved performance and fewer run-time failures.

However, we argue that such evolution often requires a giant leap, and that type inference can offer a crucial missing step. If omitted type annotations are interpreted as *unknown* types, rather than the dynamic type, then static types can often be inferred, thereby removing unnecessary assumptions of the dynamic type. The remaining assumptions of the dynamic type may then be removed by either reasoning outside the static type system, or restructuring the code.

We present a type inference algorithm that can improve the performance of existing gradually typed programs without introducing any new run-time failures. To account for dynamic typing, types that flow in to an unknown type are treated in a fundamentally different manner than types that flow out. Furthermore, in the interests of backward-compatibility, an escape analysis is conducted to decide which types are safe to infer. We have implemented our algorithm for ActionScript, and evaluated it on the SunSpider and V8 benchmark suites. We demonstrate that our algorithm can improve the performance of unannotated programs as well as recover most of the type annotations in annotated programs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Languages, Performance, Theory

Keywords Gradual typing, Type inference, ActionScript

1. Introduction

Gradual Typing and Evolutionary Programming Gradual typing [12, 17] aims to combine the benefits of static typing and dynamic typing in a language. In a gradually typed program, dynamically typed code can be mixed with statically typed code. While the dynamically typed fragments are not constrained to follow the structure enforced by a static type system, the statically typed fragments enjoy not only some static safety guarantees ("well-typed programs cannot be blamed"—the blame theorem [24]) but also admit performance optimizations that the dynamically typed fragments do not. Gradual typing envisions a style of programming where dynamically typed programs can be *evolved* into statically typed programs by gradually trading off liberties in code structure for assurances of safety and performance.

Although there has been much recent progress on mastering the recipe of gradual typing, a key ingredient has been largely missing—*type inference*. The only previous work on type inference for gradually typed languages is based on unification [18], which is unsuitable for use in object-oriented languages with subtyping. Unfortunately, as we argue below, the lack of type inference may be the most significant obstacle towards adopting the style of evolutionary programming envisioned by gradual typing.

The Key Missing Ingredient: Type Inference In a gradually typed language, a program may be partially annotated with types. Any missing types are uniformly assumed to be the dynamic type. This means that the fragments of the program that have missing type annotations do not enjoy any of the benefits of static typing. In particular, their performance is hindered by dynamic casts, even if they implicitly satisfy the constraints of static typing (i.e., even if the dynamic casts never fail). To improve performance, the missing types have to be declared.

However, in our experience with a mainstream gradually typed language, ActionScript [13], the task of evolving dynamically typed programs to statically typed programs by declaring missing types can be quite onerous. The annotation burden is often intimidating: in the limit, types must be declared for every variable in the evolving fragment, and the programmer may need to juggle several annotations to satisfy the constraints of the static type system.

Furthermore, the programmer may eventually be forced to declare dynamic types for some variables: it may not be possible to satisfy the constraints of the static type system without restructuring the code. This is because the programmer may be relying on an invariant that cannot be expressed via static types in the language, i.e., the proof of safety may rely on a form of reasoning (typically, path-sensitive) that is outside the scope of the static type system.

Unfortunately, due to these difficulties, gradually typed programs often continue to miss out on important benefits of static typing, such as performance optimizations. Consequently, evolutionary programming remains a fantasy.

The crux of the problem is that a missing type is misinterpreted as the dynamic type, whereas the intended interpretation is that of an *unknown* type. Often, an unknown type can be inferred to be a static type. In contrast, the dynamic type is a "fallback" to encode imprecision: a variable must be of the dynamic type when the set of values that the variable may denote cannot be expressed as a static type in the language, thereby forcing imprecision in the type abstraction.

Therefore, we envision an iterative process for evolution of dynamically typed programs to statically typed programs, that alternates between two states. In one state, type inference removes un-

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01...$10.00

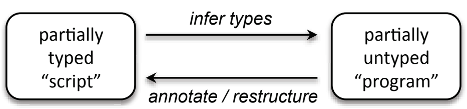

Figure 1. Iterative Evolution of Scripts to Programs

necessary assumptions of the dynamic type, so that elimination of any remaining dynamic types in the code requires either reasoning outside the static type system, or restructuring the code. In the other state, the programmer either introduces further type annotations or restructures parts of the program to conform to the static type discipline. Furthermore, at any state, the programmer may decide to pause the evolution process, deploy the program, and take advantage of the improved precision of static types that have already replaced dynamic types in the program. At a later point, the programmer should be free to resume the evolution process. This process is depicted in Figure 1.

Since the programmer may choose to evolve an existing, deployed program, type inference must ensure that the program remains backward-compatible: it should have exactly the same run-time behavior after type inference as it would have before type inference. In particular, if the program was safe (did not fail at run time) then it should continue to be safe, and any other programs that ran safely with the program should continue to run safely.

Retrofitting Type Inference on Gradually Typed Programs In this paper we study the problem of retrofitting type inference in an existing gradually typed language. The practical motivation for our work is to improve the performance of existing ActionScript programs "under the hood." Our aim is to let existing programs reap the benefits of type-directed performance optimizations as much as possible, while remaining backward-compatible.

On the other hand, our aim is not to eliminate run-time failures in existing programs: indeed, any run-time failures need to be preserved as well. Thus, the static types we infer may not satisfy some static safety guarantees, such as the blame theorem [24]. Nevertheless, the static types we infer can improve performance, since they soundly approximate precise sets of run-time values. Thus, we prefer to infer static types where possible rather than falling back on the (imprecise) dynamic type. Of course, it remains possible to recover static safety guarantees by forgoing some opportunities for optimization—we can fall back on the dynamic type whenever the inferred static type may be cast to an incompatible type.

Design Challenges The aim of our type inference algorithm is to reduce the imprecision associated with dynamic types, so that programs with missing type annotations can benefit from various performance optimizations. We try to infer, for each variable, a precise type abstraction of the set of values that flow into it. The setting of a gradually typed language and the requirement of backward-compatibility present several unique challenges, as outlined next.

In a purely static type system, the definitions of a variable determine a lower bound on its type, since the type must admit every value flowing into the variable; dually, the uses of a variable determine an upper bound on its type, since the type must admit only those values that are admitted by the type of every context to which the variable flows out. Thus, for a type that satisfies both sets of constraints, it is possible to prove statically that every use of the variable is safe for every definition of that variable. In contrast, in a gradual type system, every use of a variable need not be safe for every definition of that variable: instead, run-time failures may be avoided if each use is safe for *some* definition. Thus, the type of a variable must be inferred by considering only its definitions, deferring type checks on its uses until run time.

Furthermore, the type of a variable may only be inferred if it is *local*, i.e., if all of its definitions are visible during compilation. In the absence of this guarantee, we cannot exclude the possibility that values of an unexpected type may flow into the variable at run time: for example, consider a parameter of a global function, which can be defined by arbitrary arguments through calls that are not visible during compilation. In such cases, replacing the variable's assumed dynamic type with an inferred static type may break the program in some unknown run-time environment.

Based on these observations, we infer types of only those variables for which we can infer the types of all values flowing in, i.e., their *inflows*; and we derive the solutions by computing the least upper bounds of those types, without checking that the solutions satisfy the greatest lower bounds of the types of the contexts to which they flow out, i.e., their *outflows*.

Higher-order values, such as functions and objects, present some interesting subtleties. There is an immediate problem if we take the standard least upper bounds of their (higher-order) types, since we will then be taking the greatest lower bounds of the "input" (negative) parts of these types—which violates the principle we motivated above, giving us prescriptive "safe" types rather than descriptive "precise" types for the input parts of the higher-order solution. Another possibility might be to take the least upper bounds of both "input" and "output" parts, following naïve subtyping [24][1]. However this also is not quite right, since we are still neglecting to observe the types of the actual inputs, e.g., the types of arguments in calls to a function; rather, as with standard subtyping, we are still considering the types of values that such a function considers safe to input, i.e., the function's parameter types. The key insight is that if we are to obey the principle of solving based on inflows alone, we must *recursively deconstruct* higher-order types down to their first-order parts, solve for those based on inflows (e.g., the inflow of arguments to the parameter of a function-typed variable), and then reconstruct the higher-order parts in such a way as to preserve the desired asymmetry.

We devise an elegant way of inferring such higher-order solutions, via a notion of *kinds*; this requires us to treat higher-order inflows in a fundamentally different manner than higher-order outflows (Section 2.3).

Another complication arises when local functions and objects *escape*. In this case we cannot infer the parts of their types in negative ("input") positions, since values may then flow into them from some code that cannot be analyzed at compile time. Therefore, it seems that we need some kind of escape analysis to figure out if some local function or object escapes. Perhaps surprisingly, the escape analysis we require is carried out "for free" by our algorithm: it suffices to assume that in the enclosing program's type, the parts in negative positions are either explicitly annotated, or are the dynamic type (Section 2.5).

Guarantees Our main guarantee is that the type inference algorithm is sound: if a program fails at run time with our inferred types, it would have failed at exactly the same point with the dynamic type. The soundness guarantee is compositional, in the sense that it holds even when the program is placed in an arbitrary run-time environment. Furthermore, we show how to recover solutions from our inferred types that satisfy the blame theorem [24]. We also prove that the time complexity of our type inference algorithm is quadratic in the size of the input program.

Implementation and Evaluation We have implemented our type-inference algorithm for ActionScript, and evaluated it on the SunSpider [20] and V8 [23] benchmark suites. Overall, we are able to

[1] Naïve subtyping is covariant in both negative and positive positions, unlike standard subtyping (which is contravariant in negative positions).

achieve 1.6x performance improvement on average over partially typed benchmarks (with type annotations only on required parts of the interfaces), with a maximum performance improvement of 5.6x (Section 5).

Contributions To summarize, we make the following main contributions in this paper:

- We argue that the evolution of gradually typed programs is essentially an iterative process, with type inference being a key missing component.
- We design a type inference algorithm for gradually typed languages with the goal of improving the performance of programs as much as possible, while preserving the run-time behaviors of unannotated programs. Our design involves new ideas that are specific to the setting of a gradual type system and the requirement of backward-compatibility, such as the need to treat definitions and uses asymmetrically, and the need for escape analysis to decide what types are safe to infer.
- We formalize our type inference algorithm in a core calculus of functions and objects, and prove soundness and complexity theorems for the algorithm. Technical novelties in our algorithm include a notion of kinds to compute precise higher-order solutions, and the encoding of escape analysis as a syntactic condition on the program's type.
- We implement our algorithm for ActionScript, a mainstream gradually typed language, and evaluate its effectiveness, with encouraging results.

Overall, we believe that the techniques and insights developed in this paper will elucidate the intricacies of type inference for gradually typed languages, and spur further research in this area.

2. Overview

2.1 Algorithm

While existing compilers for gradually typed languages such as ActionScript [13] consider missing type annotations to be the dynamic type, our aim is to infer precise static types where possible, and fall back on the dynamic type only where necessary.

As usual, our type inference algorithm begins by replacing missing type annotations in the program with fresh *type variables*. Then we compile the program, using type annotations to generate *coercions* between types.

A coercion is of the form $S \rhd T$, where S and T are types; such a coercion witnesses the flow of a term of type S into a context of type T. In particular, coercions involving type variables are interpreted as *flows* for those type variables: for example, if X is a type variable, then $T \rhd X$ is an *inflow* for X, whereas $X \rhd T$—where T is not a type variable—is an *outflow* for X.

Starting from the original set of coercions collected during compilation, we now iterate, generating new coercions from existing ones using carefully designed rules which treat the implications of inflows and outflows differently. At each step, new coercions represent flows that are implied by, but are not directly expressed by, extant coercions.

Iteration continues until our rules can produce no new coercions, at which point we have a *closure* of the flows implied by the type annotations in the program. For each type variable X, we can now derive a solution, namely the least upper bound of all types T that can flow into X—via inflows expressed by coercions of the form $T \rhd X$. A formal treatment of the algorithm is presented in Section 3.

We now demonstrate the fine points of the algorithm, viz. the computation of closures and the derivation of solutions, using examples in ActionScript [13]; ActionScript is an extension of JavaScript with a gradual type system, and is the programming language underlying Flash applications.

We show type variables as part of programs below, although they are not part of ActionScript. In actual programs, these type variables are just missing annotations. We follow the convention that type variables (in italics) have similar names as corresponding unannotated variables, with the first letter capitalized. For example, the type variable for an unannotated variable `foo` in the program is written as *Foo*. If `foo` is an unannotated function, *Foo?* represents the type variable for the parameter type of `foo` and *Foo!* represents the type variable for the return type of `foo`. The dynamic type is denoted by $\star$. `Number` and `Boolean` are base types. We also use $\bot$ to denote the bottom type, although it is not part of ActionScript. For the purpose of our examples, operators are typed as follows:

```
+ : (Number, Number) -> Number
< : (Number, Number) -> Boolean
```

2.2 Local Variables having Base Types

We start with a simple example to show how our algorithm infers primitive types for local variables in a function, and how it improves performance.

```
function foo(n:Number):Foo! {
    var index:Index = 0;
    var sum:Sum = index;
    while(index < n)
        { index = index + 1; sum = sum + index }
    return sum
}
```

In the absence of type inference, `index` and `sum` will be given type $\star$. Since the `+` operator has type `(Number, Number)` $\rightarrow$ `Number`, evaluation of `index + 1` and `sum + index` will cause in run-time conversions $\star \rhd$ `Number` for `index` and `sum`. To store the results back to `index` and `sum`, further run-time conversions `Number` $\rhd \star$ will happen. As we show later in our experiments, such run-time conversions can hurt the performance of programs.

We now show how our inference algorithm works. After adding type variables for missing types, as shown, we compile the program and collect coercions. The initializations of `index` and `sum` on lines 2 and 3 generate `Number` $\rhd$ *Index* and *Index* $\rhd$ *Sum*. The operations `index < n`, `index = index + 1`, and `sum = sum + index` on lines 4 and 5 generate *Index* $\rhd$ `Number`, `Number` $\rhd$ *Index*, *Sum* $\rhd$ `Number`, *Index* $\rhd$ `Number`, and `Number` $\rhd$ *Sum*.

To solve these flows, we note that the only type flowing into *Index* is `Number`. So, we infer *Index* = `Number`. Next, we see that types flowing into *Sum* are `Number` and *Index*. Therefore, the solution for *Sum* is the union of `Number` and *Index*, which is `Number`. And finally, the only type flowing into *Foo!* is *Sum*. So, *Foo!* = `Number` as well.

Annotating `index` and `sum` with `Number` rather than $\star$ improves run-time performance by eliminating unnecessary run-time conversions between `Number` and $\star$ in the loop body.

Note that we also generated *Sum* $\rhd$ `Number` and *Index* $\rhd$ `Number` but did not use them for inference. A typical type checker will check that the solutions of *Sum* and *Index* satisfy these outflows, but we skip this step. We say more about why we do so in Section 2.6.

2.3 Local Variables having Function Types

2.3.1 Unions of Function Types are Problematic

When a base type (like `Number` or `Boolean`) flows into a type variable, the solution of the type variable is simply that type; and if several base types flow in, the solution is the union of those base types. Unfortunately, the obvious ways of generalizing this approach to higher-order types (like function types) do not work.

For example, consider the following program:

```
8  var x:X = function(a:*):Number { ... };
9  x(1)
```

On line 8, we generate $\star \rightarrow \texttt{Number} \rhd X$. Thus, we may infer the solution of X to be the type flowing into it, $\star \rightarrow \texttt{Number}$. However, a more precise solution is $\texttt{Number} \rightarrow \texttt{Number}$, based on the fact that the only values flowing into the parameter of the function stored in `x` are of type `Number` (line 9).

Furthermore, when several function types flow into a type variable, we cannot combine those inflows by taking the union of all such types. Consider the following example:

```
10 var x:X;
11 if(b) { x = function(y:Number):Number { ... }; x(1) }
12 else { x = function(y:Boolean):Number { ... }; x(true) }
```

We generate $\texttt{Number} \rightarrow \texttt{Number} \rhd X$ from line 11 and $\texttt{Boolean} \rightarrow \texttt{Number} \rhd X$ from line 12. Since the union of two function types is a type $S \rightarrow T$ where S is the intersection of their parameter types and T is the union of their return types, we would infer X to be $\bot \rightarrow \texttt{Number}$ using the standard approach. This would mean that in the running program, any application of `x` to a `Number` or a `Boolean` value would result in an error. This is clearly unsound as the programmer applies `x` to a `Number` at line 11, and to a `Boolean` at line 12, both of which are safe.

2.3.2 Kinds as Higher-Order Solutions

To deal with higher-order types, we introduce a notion of *kinds*. A kind can be thought of as a "view" of a type variable: it describes a structure, and introduces type variables for the types of parts within that structure. Coercions between type variables and kinds are generated during compilation and closure computation, in either direction: a coercion from a type variable T to a kind K witnesses the flow of T-typed values into a context whose higher-order type is exemplified by K; conversely, a coercion from K to T witnesses the flow of higher-order values of kind K into a T-typed context.

For example, on line 9, the function call `x(1)` induces a view of X as a function type. To realize this view, we generate the coercion $X \rhd X? \rightarrow X!$, where $X? \rightarrow X!$ is a kind for X corresponding to function types; this witnesses values of type X flowing to a context in which they are used as functions. We also generate $\texttt{Number} \rhd X?$, which witnesses the flow of arguments of type `Number` into the parameters of those functions.

Combining $\star \rightarrow \texttt{Number} \rhd X$ and $X \rhd X? \rightarrow X!$, we get $\star \rightarrow \texttt{Number} \rhd X? \rightarrow X!$. Deconstructing this flow using contravariance for the parameter type and covariance for the return type, we get $X? \rhd \star$ and $\texttt{Number} \rhd X!$. Intuitively, these coercions witness the flows into and out of the concrete function value assigned to `x` on line 8, resulting from the call to `x` on line 9.

Solving these flows, we obtain $X? = \texttt{Number}$, and $X! = \texttt{Number}$. But what should X's solution be? We claim it should be X's function kind, i.e. $X? \rightarrow X!$, which after substituting for $X?$ and $X!$, becomes $\texttt{Number} \rightarrow \texttt{Number}$. Our closure rules introduce the necessary intermediate step, the coercion $X? \rightarrow X! \rhd X$, as a consequence of the assignment on line 8.

Thus, a kind not only gives a view of a type variable as a higher-order type, but also encodes the corresponding higher-order solution of the type variable.

In other programs, there may be no occasion to generate a coercion from a type variable X to a kind at all, due to an absence of contexts in which X-typed values are used in a syntactically higher-order way, such as the function application term `x(1)` in the previous example. Consider the following variation:

```
13 var x:X = function(a:*):Number { ... };
14 function f(y:Number -> Number):Number { y(1) }
15 f(x)
```

We generate $\star \rightarrow \texttt{Number} \rhd X$ and $X \rhd \texttt{Number} \rightarrow \texttt{Number}$ on lines 13 and 15 respectively. Here we don't need the view of X as a function type since `x` is not applied directly, but we still need to represent the solution for X. This is accomplished by the coercions introduced by our closure rules as a result of the assignment on line 13, namely $\star \rightarrow \texttt{Number} \rhd X? \rightarrow X!$, and $X? \rightarrow X! \rhd X$. Combining $X? \rightarrow X! \rhd X$ and $X \rhd \texttt{Number} \rightarrow \texttt{Number}$ gives us $X? \rightarrow X! \rhd \texttt{Number} \rightarrow \texttt{Number}$. Solving the flows gives us $X? = \texttt{Number}$ and $X! = \texttt{Number}$. As before, since only function types flow into X, its solution is its function kind $X? \rightarrow X!$, which after substituting for $X?$ and $X!$ gives us $\texttt{Number} \rightarrow \texttt{Number}$.

Finally, decomposition using kinds enables us to take unions of kinds, not types, giving us types inferred from inflows alone. For example, the decomposition of X into $X? \rightarrow X!$ lets us capture the flow of argument types to $X?$ in the calls to `x` on lines 11 and 12, yielding $\mathit{Number} \rhd X?$ and $\mathit{Boolean} \rhd X?$, respectively. These first-order flows may then be combined by taking the least upper bound in the usual way, after which reconstruction of X proceeds as described above.

2.4 Function Parameters

2.4.1 Functions Callable by Existing Code Are Problematic

Since we cannot assume that all existing ActionScript programs will be available for our analysis, a key goal of our algorithm is to ensure that our inferred types are backward-compatible—i.e., programs compiled using inferred types are able to interoperate with programs compiled under the assumption that missing type annotations are the dynamic type, without introducing any new run-time failures.

We now show an example in which we try to extend our algorithm to infer the parameter type of a function that can be called by other programs that are not available for analysis.

```
16 function foo(x:Foo?):Foo!
17     { if(b) { return x + 1 } else { return 0 } }
18 foo(1)
```

The function call `foo(1)` on line 18 generates $\texttt{Number} \rhd \mathit{Foo?}$. Furthermore, as before we generate $\mathit{Foo?} \rhd \texttt{Number}$ and $\texttt{Number} \rhd \mathit{Foo!}$ on line 17. Solving for type variables, we get $\mathit{Foo?} = \texttt{Number}$ and $\mathit{Foo!} = \texttt{Number}$. (As before, we do not use $\mathit{Foo?} \rhd \texttt{Number}$ for inference.)

Suppose that the function `foo` is part of the interface of some library, and clients of `foo` have been compiled under the assumption that *Foo?* and *Foo!* are $\star$. Then the solution for *Foo!* is sound, because it can be used in any context of type $\star$: in particular, it will not throw an error unless it flows to a context whose type is incompatible with `Number`, in which case it would already throw an error without type inference.

Unfortunately, the solution for *Foo?* is not sound. For example, suppose a caller sets the global variable `b = false` and then calls `foo` with `x = true`. In the absence of type inference, the `true` argument will be successfully converted at run time from type `Boolean` to type $\star$—after which it will go unused, since `b = false`, avoiding a run-time error. In contrast, if we add the annotation `x:Number` after type inference, this will have the effect of requiring a run-time conversion of the `true` argument from `Boolean` to `Number` at the point of the call to `foo`, and this conversion will always fail. Thus inferring $\mathit{Foo?} = \texttt{Number}$ can break existing programs.

In general, we conclude that we cannot safely infer the parameter types of functions that are part of the interface of the program with existing code, because our analysis cannot be guaranteed to include every call to such functions.

2.4.2 Local Functions

On the other hand, it is safe to infer the parameter types of *local* functions—i.e., functions defined within other functions and objects—whenever all their calls are available for analysis.

Consider the local function `foo` in the following example:

```
19 function bar(y:Number):Bar! {
20     function foo(x:Foo?):Foo!
21         { if(b) { return x + 1 } else { return 0 } }
22     return foo(y)
23 }
```

As before, we get the solutions *Foo?* = `Number` and *Foo!* = `Number`. (Here, we also get *Bar!* = `Number`.) Since `foo` is a local function that does not *escape* `bar`, we know it cannot be called from outside `bar`, meaning that all calls to `foo` are available for analysis. In this case it is safe to infer *Foo?* = `Number`, even if the function `bar` is callable by existing code.

2.4.3 Local Functions that Escape Are Problematic

However, if a local function escapes—e.g. by being returned as a higher-order value from a function that is callable by existing code—then it becomes available to callers in the wild. This again makes it unsafe to infer the local function's parameter type, since not every call to the function is available for analysis.

```
24 function bar(y:Number):Bar! {
25     function foo(x:Foo?):Foo!
26         { if(b) { return x + 1 } else { return 0 } }
27     foo(y);
28     return foo
29 }
```

In this example, inferring *Foo?* = `Number` based on function call `foo(y)` on line 27 would again be unsound, assuming the function `bar` is callable by existing code. Since `foo` is returned by `bar`, clients can set `b = false` and call `foo` with any `x` value. Thus, as in Section 2.4.1, annotating `x` as `Number` could introduce run-time errors in existing programs.

2.5 What Types are Safe To Infer?

To summarize, the examples above illustrate that while it may be safe to infer the types of local variables in a function, the return types of functions, and the parameter types of local functions that do not escape, it is definitely not safe to infer parameter types of functions that are callable by existing code, and those of local functions that do escape. Furthermore, if values from such parameters flow into other variables, then inferring the types of those variables is clearly also unsafe.

So what is the principle that lets us decide which types are safe to infer?

2.5.1 Seeing All Inflows is Necessary

We observe that in a gradually typed language, we must see the types of all the values that may flow into a variable before we can infer the variable's type.

Suppose that at compile time we see only a proper subset of the types of values that may flow into a variable x—i.e., some of the types of values that may flow into x are unknown at compile time, because they flow into x from a run-time environment that cannot be analyzed during compilation.

Next, suppose that the compiled program is executed in such a run-time environment. In the absence of type inference, x has type $\star$, which means that the run-time environment could write any type of value into x.

Furthermore, the programmer could reason outside the type system to ensure that once such a value reaches x, it is then used correctly under appropriate conditions.

Thus, if the inferred type of x is based only on the subset of types that are seen, then we could end up breaking the program in this run-time environment, by causing a run-time error to be thrown upon the write to x: not only would this error not have occurred in the absence of type inference, but as described in the previous paragraph, it is quite possible that no error would have occurred at all.

2.5.2 Flows Encode Escape Analysis

For a function that is callable by existing code, we require the programmer to explicitly annotate the parts of the function's type that have negative polarity (in particular, the parameter type), if those parts have static types. In the absence of annotations, we must assume they have the dynamic type. For a local function, our ability to infer the parts of its type with negative polarity depends on whether the function escapes into an unknown run-time environment.

There are many ways a local function can escape: it can be returned as a value, it can be assigned to a global variable, it can be assigned to some object property and that object could be returned, and so on. At first glance, it seems that either we need a sophisticated escape analysis to figure out if a local function escapes, or we need the programmer to annotate parameter types for local functions as well.

Fortunately, we observe that the flows already encode an escape analysis. If we require that types with negative polarities in the interface of the program with existing code be explicitly annotated if static—i.e., if we assume that they are the dynamic type when unannotated—then our closure computation ensures that escape information is properly propagated to all type variables.

Let us consider the example from Section 2.4.3 with different possibilities for *Bar!*.

First, suppose that the programmer has annotated the return type of `bar` as `Number` $\rightarrow$ `Number`. In this case, the programmer has annotated the types with negative polarity in `bar` explicitly. Now, it's safe to infer *Foo?* = `Number`, since the programmer has told us that `foo` will always be called with `x` as `Number` (otherwise there will be a run-time error).

On the other hand, if the programmer had not explicitly annotated the return type of `bar`, we would have assumed it to be some type variable *Bar!*. As before, we generate *Foo?* $\rightarrow$ *Foo!* $\triangleright$ *Bar!* from `return foo` on line 28. We introduce function kind (Section 2.3.2) for *Bar!* and add *Foo?* $\rightarrow$ *Foo!* $\triangleright$ *Bar!?* $\rightarrow$ *Bar!!* and *Bar!?* $\rightarrow$ *Bar!!* $\triangleright$ *Bar!*. Now, we observe that *Bar!?* has negative polarity. Since the function `bar` is callable by existing code, we assign *Bar!?* = $\star$. Thus, we have *Foo?* $\rightarrow$ *Foo!* $\triangleright$ $\star$ $\rightarrow$ *Bar!!*. Deconstructing the flow, we get $\star \triangleright$ *Foo?*. And finally, we infer *Foo?* = $\star$, which is always safe.

In this way, escape analysis is manifested through our closure computation. Thus, we require explicit type annotations only for types that have negative polarities in the interface of the program with existing code; all the remaining types we can infer.

2.6 Outflows

So far we have maintained that we are looking at inflows, of the form $T \triangleright X$, to solve for type variable X. But we have not said anything about outflows, of the form $X \triangleright T$, where T is not a type variable. Such outflows correspond to dynamic consistency checks.

2.6.1 Outflows Need Not Contribute to Solutions

We observe that outflows, of the form $X \triangleright T$, need not contribute to the solution for X—although in general, such outflows do play a role in propagating flows between parts of higher-order types, as we have seen in Section 2.3.2. If we can analyze all writes to X, we know precisely the types of all values that X will need to hold at run time and so, by just looking at inflows, of the form $T \rhd X$, we can infer a more precise solution for X.

2.6.2 Outflows Need Not be Validated Statically

We also do not need to validate the outflows at compile time, as the runtime does it anyway. As we show in the following example, if we try to validate outflows statically, we could end up being too conservative and infer less precise types than we could.

```
30 function foo(f:Foo?):Number {
31     if(b) { return f(true) } else { return 0 }
32 }
33 if(b) { foo(function (x:Boolean):Number { ... }) }
34 else { foo(function (x:Number):Number { ... }) }
```

First we note that this program is safe if *Foo?* = $\star$, which is the case when there is no type inference. When `b` is `false`, `foo` doesn't use its argument and returns `0`.

In our inference algorithm, we generate the flows: `Boolean` $\rightarrow$ `Number` $\rhd$ *Foo??* $\rightarrow$ *Foo?!* (line 33), `Number` $\rightarrow$ `Number` $\rhd$ *Foo??* $\rightarrow$ *Foo?!* (line 34), and `Boolean` $\rhd$ *Foo??* (from function call `g(true)` on line 31). Deconstructing the first two flows gives us *Foo??* $\rhd$ `Boolean`, *Foo??* $\rhd$ `Number`, and `Number` $\rhd$ *Foo?!*. Therefore, we infer *Foo??* = `Boolean` and *Foo?!* = `Number`, and our solution is *Foo?* = `Boolean` $\rightarrow$ `Number`. With these inferred types, the program is still safe. There is no change in the run-time semantics of the program.

But note that our solution *Foo??* = `Boolean` turns the outflow *Foo??* $\rhd$ `Number` to the dynamic consistency check `Boolean` $\rhd$ `Number`, which will fail, if executed at run time. So, if we were to validate outflows statically, we would have promoted *Foo??* to $\star$, which means we would have inferred a less precise type that we could.

In this example, the programmer encodes path sensitive reasoning about correctness of his program and ensures that the dynamic consistency check `Boolean` $\rhd$ `Number` never happens at run time (when the function in line 34 is passed to `foo`, `foo` doesn't invoke it). Since it is not possible to statically determine whether dynamic consistency checks will fire at run time, we do not validate outflows. But, we maintain that our inference doesn't introduce any new run-time errors. Our soundness theorem proves that if a program fails at run time with our inferred types, it would have failed even if those types were $\star$, and vice versa.

2.6.3 Validating Outflows for Blame Guarantees

Traditionally, type inference algorithms in statically typed languages ensure that the solution of X is a subtype of the greatest lower bound of all the types T that flow out of X. This implies that all dynamic consistency checks always succeed.

In our system we do not validate outflows because this may unnecessarily constrain our inferred types, as we saw in the previous example. But this comes with a trade-off. Type inference for statically typed languages guarantees that inferred types will not generate any run-time type errors. While we clearly cannot hope for such guarantees, gradually typed languages often provide weaker blame guarantees [24]—the statically typed parts of a program cannot be blamed for any run-time type errors. On the other hand, by default, we do not give any such guarantee: dynamic consistency checks may throw errors at run time. Instead, our inferred types can be thought of as directives to a runtime to optimize memory, run-time conversions, property lookups, and so on; we only guarantee that they do not introduce any new run-time errors in the program.

If, on the other hand, the programmer wants some blame guarantees from our system, we can run an additional step wherein we validate outflows. If some outflow may cause an error, we either promote the type variable that may cause the error to the $\star$ type, or if there is no type variable involved, fail the compilation. In the example of previous section, once we find that inferring *Foo??* = `Boolean` will fail the consistency check *Foo??* $\rhd$ `Number`, we can promote the type variable *Foo??* = $\star$, and recover blame guarantees for our system.

2.7 Objects

The treatment of objects in our system follows that of functions, in that we use polarities to model object properties with various access controls. For example, private properties of objects have no polarity, so we infer types of private properties. Read-only properties, like public functions, have positive polarity and so, we infer positive parts of the types of those functions and we require programmers to annotate the negative parts. On the other hand, public variables in an object are read-write, so they have both negative and positive polarities, and we require programmers to annotate the types of public variables. As before, we do not need escape analysis to figure out if some private function escapes the scope of an object, or if a private object escapes the scope of a function. Escape analysis is manifested in the closure computation. We show an example in ActionScript where object templates are defined using classes.

```
class A {
    private var b:B = true;
    private function foo(var x:Foo?):Foo!
        { if(b) { return x + 1; } else { return 0; } }
    public function bar():Bar! { return foo; }
}
```

In this example, our solution will be: *B* = `Boolean`, *Foo?* = $\star$, *Foo!* = `Number`, *Bar!* = $\star$ $\rightarrow$`Number`. Note that as before, we infer *Foo?* = $\star$ because it is returned from a public function `bar`.

Summary Our key observations can be summarized as follows.

1. Unlike type inference in statically typed languages, we treat inflows, which represent definitions, in a fundamentally different manner than outflows, which represent uses. The inflows for a type variable determine its solution. The outflows for a type variable represent dynamic consistency checks that happen at the runtime.
2. When the inflows for a type variable involve higher-order types, its solution is encoded by a kind, which deconstructs the the type variable into parts, recursively solves for those parts, and reconstructs the solutions to determine the solution of the type variable. In particular, the negative parts of such a solution are determined by the negative parts of higher-order types in the outflows for the type variable.
3. We need to see all the types of all values flowing into a variable before we can infer its type. This means that we can only infer positive parts of the type that serves as the program's interface with existing code. The negative parts of that interface can be defined by some unknown run-time environment, so we either need the programmer to annotate those parts explicitly, or we assume them to be the dynamic type.
4. We do not need a separate escape analysis to find which types escape their scope. Once we have put sufficient type annotations in the interface of the program with existing code, our flows encode the escape analysis.
5. Our framework extends naturally to objects (and classes). We use polarities to model object properties with various access controls: private, read-only, write-only, and public.

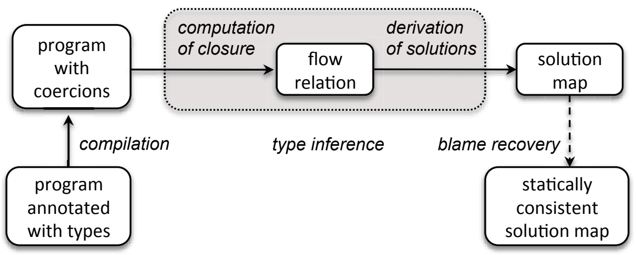

Figure 2. Overall Architecture for Type Inference

3. Formal Development

In this section, we formally develop our type inference algorithm for a core gradually typed calculus of functions and objects.

Unknown types are modeled as type variables in the language. We formalize the static semantics of the language by a compilation judgment that translates a program annotated with types to a program with coercions (Section 3.2). We then describe a type inference algorithm that analyzes the coercions to compute a flow relation over types, and derives a solution map for the type variables based on this flow relation (Section 3.3). We also describe how the solution map may be "weakened" to recover standard blame guarantees (Section 3.4). Figure 2 depicts the overall architecture for type inference.

We prove complexity theorems for the type inference algorithm (Section 3.5). Next, we formalize the dynamic semantics of the language by an evaluation judgment that, under a solution map for the type variables, reduces programs with coercions to values (Section 3.6). Based on this dynamic semantics, we prove soundness theorems for the type inference algorithm, by comparing the behaviors of programs under the solution map inferred by our algorithm with those under the "default" solution map that interprets every type variable as the dynamic type (Section 3.7).

3.1 Syntax

The syntax of the language is shown below. We model function parameters and object properties as variables x. In contrast to standard object calculi, we distinguish readability and writability of object properties: we model access capabilities κ for object properties as subsets of $\{+, -\}$, where $+$ denotes readability and $-$ denotes writability. We express objects and their types as indexed collections of object properties and their types; for any non-negative integer m the notation $[m]$ means the set of indices $\{1, \ldots, m\}$, and for any syntactic entity ϕ the notation $\{\phi_i\}^{i\in[m]}$ means the collection $\{\phi_1, \ldots, \phi_m\}$ indexed by $[m]$.

$$\begin{array}{rrcl}
\text{term} & t & ::= & \mathsf{null} \\
 & & \mid & \mathsf{fun}\ (x\!:\!T)\ t\!:\!T' \mid \{x_i^{\kappa_i}\!:\!T_i = t_i\}^{i\in[m]} \\
 & & \mid & x \mid x = t \\
 & & \mid & t(t') \mid t.x \mid t.x = t' \\
 & & \mid & \langle T \rhd T'\rangle\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' \\
\text{type variable} & X & ::= & \alpha \mid X? \mid X! \mid X.x \\
\text{type} & T & ::= & \bot \\
 & & \mid & T \to T' \mid \{x_i^{\kappa_i} : T_i\}^{i\in[m]} \\
 & & \mid & \star \mid X
\end{array}$$

Terms include functions (of the form $\mathsf{fun}\ (x : T)\ t : T'$), objects (of the form $\{x_i^{\kappa_i} : T_i = t_i\}^{i\in[m]}$), and the null value (null), as well as applications of functions (of the form $t(t')$), reads and writes of object properties (of the form $t.x$ and $t.x = t'$), and null checks (of the form $\mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t''$). They also include reads and writes of variables (of the form x and $x = t$). Furthermore, they include coercions between types (of the form $T \rhd T'$). Coercions may be interpreted syntactically as flows, and semantically as representation-conversion operations. We assume that in source code, there are no coercions.

Types include function types (of the form $T \to T'$), objects types (of the form $\{x_i^{\kappa_i} : T_i\}^{i\in[m]}$), the null type ($\bot$), and the dynamic type ($\star$). Furthermore, they include type variables X. Some type variables may be derived by others: $X.x$ denotes the type of the property x of objects of type X, while $X?$ and $X!$ denote the type of the input and output of functions of type X. We assume that in source code, type variables are distinct and are of the form α.

Recursive functions, self references, and loops can be readily encoded in the language; so can blocks with "local" variables. Thus, we do not include those constructs in the language. Furthermore, we do not model classes: their treatment is similar to that of objects, and we discuss them further in Section 4.

3.2 Compilation

We now describe the compilation semantics of our language. The goal of compilation is to compute types and embed coercions in source programs, thereby preparing them for type inference (Section 3.3) and evaluation (Section 3.6).

Compilation proceeds under a type environment. A type environment Γ is a finite map from variables to types. Compilation judgments are of the form $\Gamma \vdash t \hookrightarrow t' :: T$, meaning that t compiles to t' of type T under Γ. Figure 3 lists the rules for deriving compilation judgments.

The compilation rules are fairly straightforward: the type of a term is computed in a syntax-directed manner, and whenever a term of type T appears in a context that is annotated with type T', the coercion $T \rhd T'$ is attached to the term. We elaborate on the rules for deconstructing functions and objects, namely (C-APP) for function applications and (C-PROPR) and (C-PROPW) for object property accesses: they rely on a partial relation $\multimap$ between types, defined below. Intuitively, this relation captures the condition under which a term of a certain type can be deconstructed, by viewing the type either as the type of a function that can be applied or as the type of an object for which a property can be read or written.

Definition 3.1 (View). *A type T can be viewed as a type T' if $T \multimap T'$ can be derived by any of the following rules.*

- $X \multimap X? \to X!$
- $\star \multimap \star \to \star$
- $T \to T' \multimap T \to T'$
- $X \multimap \{x^\kappa : X.x\}$ *for $\kappa = \{+\}$ or $\kappa = \{-\}$*
- $\star \multimap \{x^\kappa : \star\}$ *for $\kappa = \{+\}$ or $\kappa = \{-\}$*
- $\{x_i^{\kappa_i} : T_i\}^{i\in[m]} \multimap \{x_j^\kappa : T_j\}$ *such that $j \in [m]$ and $\kappa \subseteq \kappa_j$, for $\kappa = \{+\}$ or $\kappa = \{-\}$*

Compilation may fail due to the partiality of the view relation: essentially, whenever a term of some type is deconstructed in a context that is annotated with an "incompatible" type. In particular, a function cannot be viewed as an object, an object cannot be viewed as a function, and an object without a particular access capability for a property cannot be viewed as an object with that access capability for that property.

Further restrictions can be imposed statically. In particular, a standard gradual type system would detect other "unsafe" coercions, by interpreting all type variables as the dynamic type $\star$ and ensuring that coercions are between *statically consistent* types (cf. consistency and consistent-subtyping [16, 17]). Static consistency is defined in Section 3.4, where we discuss how to carry over "blame" guarantees offered by such systems to our setting. How-

Compilation judgment $\Gamma \vdash t \hookrightarrow t' :: T$

(C-NULL)

$$\frac{}{\Gamma \vdash \mathsf{null} \hookrightarrow \mathsf{null} :: \bot}$$

(C-FUN)

$$\frac{\Gamma[x \mapsto T_1] \vdash t_2 \hookrightarrow t_2' :: T_2' \qquad t_2'' = \langle T_2' \rhd T_2 \rangle\, t_2'}{\Gamma \vdash \mathsf{fun}\ (x\!:\!T_1)\ t_2\!:\!T_2 \hookrightarrow \mathsf{fun}\ (x\!:\!T_1)\ (t_2'')\!:\!T_2 :: T_1 \to T_2}$$

(C-OBJ)

$$\frac{\begin{array}{c}\Gamma' = \Gamma[x_1 \mapsto T_1, \ldots, x_m \mapsto T_m] \\ \forall i \in [m].\quad \Gamma' \vdash t_i \hookrightarrow t_i' :: T_i' \quad t_i'' = \langle T_i' \rhd T_i \rangle\, t_i'\end{array}}{\Gamma \vdash \{x_i^{\kappa_i}\!:\!T_i = t_i\}^{i \in [m]} \hookrightarrow \{x_i^{\kappa_i}\!:\!T_i = t_i''\}^{i \in [m]} :: \{x_i^{\kappa_i}\!:\!T_i\}^{i \in [m]}}$$

(C-PROPR)

$$\frac{\Gamma \vdash t \hookrightarrow t' :: T \qquad T \multimap \{x_j^{\{+\}} : T_j\}}{\Gamma \vdash t.x_j \hookrightarrow (\langle T \rhd \{x_j^{\{+\}} : T_j\} \rangle\, t').x_j :: T_j}$$

(C-PROPW)

$$\frac{\begin{array}{c}\Gamma \vdash t \hookrightarrow t' :: T \qquad T \multimap \{x_j^{\{-\}} : T_j\} \\ \Gamma \vdash t_j \hookrightarrow t_j' :: T_j' \qquad t_j'' = \langle T_j' \rhd T_j \rangle\, t_j'\end{array}}{\Gamma \vdash t.x_j = t_j \hookrightarrow (\langle T \rhd \{x_j^{\{-\}} : T_j\} \rangle\, t').x_j = t_j'' :: T_j}$$

(C-APP)

$$\frac{\begin{array}{c}\Gamma \vdash t \hookrightarrow t' :: T \qquad T \multimap T_1 \to T_2 \\ \Gamma \vdash t_1 \hookrightarrow t_1' :: T_1' \qquad t_1'' = \langle T_1' \rhd T_1 \rangle\, t_1'\end{array}}{\Gamma \vdash t\ (t_1) \hookrightarrow (\langle T \rhd T_1 \to T_2 \rangle\, t')\ (t_1'') :: T_2}$$

(C-VARR)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \hookrightarrow x :: T}$$

(C-VARW)

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash t \hookrightarrow t' :: T'}{\Gamma \vdash x = t \hookrightarrow x = \langle T' \rhd T \rangle\, t' :: T}$$

(C-IF)

$$\frac{\begin{array}{c}\Gamma \vdash t \hookrightarrow t' :: T \\ \alpha \text{ fresh} \qquad \forall i \in \{1,2\}.\quad \Gamma \vdash t_i \hookrightarrow t_i' :: T_i \quad t_i'' = \langle T_i \rhd \alpha \rangle\, t_i'\end{array}}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \hookrightarrow \mathsf{if}\ t'\ \mathsf{then}\ t_1''\ \mathsf{else}\ t_2'' :: \alpha}$$

Figure 3. Compilation

ever, we emphasize that we do not require those restrictions for our soundness theorem: we can tolerate existing run-time failures as long as we do not introduce any new run-time failures.

3.3 Type Inference

Following compilation, the coercions embedded in a program say what types of terms appear in what types of contexts. Other coercions are implicit in the type of the compiled program, which serves as the interface with existing code: these are coercions between the dynamic type and type variables in the type of the compiled program (which continue to be interpreted as the dynamic type by existing code). Based on this set of coercions, we conduct type inference as follows:

1. We apply some carefully designed closure rules on this set of coercions, to compute a flow relation between types (Section 3.3.1). The flow relation overapproximates the set of all possible flows of terms to contexts at run time.
2. We then derive a solution map for type variables based on that flow relation (Section 3.3.2). In particular, if some context is annotated by a type variable, the solution for that type variable is an overapproximation of the types of terms that may flow to that context at run time.

Flow judgment $T \rhd T' \checkmark$

(F-BASE)

$$\frac{T \rhd T' \text{ is a coercion in the compiled program}}{T \rhd T' \checkmark}$$

(F-COMP)

$$\frac{\begin{array}{c}T \text{ is the type of the compiled program} \\ C \in \{X \rhd \star \mid X \text{ is positive in } T\} \cup \{\star \rhd X \mid X \text{ is negative in } T\}\end{array}}{C \checkmark}$$

(F-PULL)

$$\frac{\mathbb{K}^X \rhd X \checkmark \qquad X \rhd Y \checkmark}{\mathbb{K}^X \rhd Y \checkmark}$$

(F-FACTOR)

$$\frac{T \rhd X \checkmark \qquad \|T\|_X = \mathbb{K}^X \qquad C \in \{T \rhd \mathbb{K}^X, \mathbb{K}^X \rhd X\}}{C \checkmark}$$

(F-TRAN)

$$\frac{\mathbb{K}^X \rhd X \checkmark \qquad X \rhd T \checkmark \qquad \mathbb{K}^X \preccurlyeq T}{\mathbb{K}^X \rhd T \checkmark}$$

(F-EXPFUNL)

$$\frac{\star \rhd T_1 \to T_2 \checkmark}{\star \to \star \rhd T_1 \to T_2 \checkmark}$$

(F-EXPOBJL)

$$\frac{\star \rhd \{x_i^{\kappa_i} : T_i\}^{i \in [m]} \checkmark}{\{x_i^{\kappa_i} : \star\}^{i \in [m]} \rhd \{x_i^{\kappa_i} : T_i\}^{i \in [m]} \checkmark}$$

(F-EXPFUNR)

$$\frac{T_1 \to T_2 \rhd \star \checkmark}{T_1 \to T_2 \rhd \star \to \star \checkmark}$$

(F-EXPOBJR)

$$\frac{\{x_i^{\kappa_i} : T_i\}^{i \in [m]} \rhd \star \checkmark}{\{x_i^{\kappa_i} : T_i\}^{i \in [m]} \rhd \{x_i^{\kappa_i} : \star\}^{i \in [m]} \checkmark}$$

(F-SPLITFUN)

$$\frac{T_1 \to T_2 \rhd T_1' \to T_2' \checkmark \qquad C \in \{T_1' \rhd T_1, T_2 \rhd T_2'\}}{C \checkmark}$$

(F-SPLITOBJ)

$$\frac{\begin{array}{c}\{x_i^{\kappa_i} : T_i\}^{i \in [m]} \rhd \{x_i^{\kappa_i'} : T_i'\}^{i \in [n]} \checkmark \\ n \le m \qquad \forall i \in [n] : \kappa_i' \subseteq \kappa_i \\ j \in [n] \qquad C \in \{T_j \rhd T_j' \mid + \in \kappa_j'\} \cup \{T_j' \rhd T_j \mid - \in \kappa_j'\}\end{array}}{C \checkmark}$$

Figure 4. Computation of Closure

3.3.1 Computation of Closure

Given a compiled program and its type, we compute a flow relation between types by initializing and growing a set of coercions until fixpoint. Flow judgments are of the form $T \rhd T' \checkmark$, meaning that "flows" from type T to type T' are possible, i.e., terms of type T may flow to contexts annotated with type T' at run time. Figure 4 lists the rules for deriving flow judgments. Those rules are explained below.

Unlike usual flow relations for statically typed languages with subtyping, our flow relation is not merely a transitive closure of the coercions in a compiled program, interpreted as subtyping constraints. Instead, it is designed carefully to account for gradual typing, and is tuned for efficiency.

Rules (F-BASE) and (F-COMP) generate the initial facts. In particular, (F-COMP) relies on a standard notion of polarity for type variables, defined below, to ensure that any type variable that appears in the type of the compiled program is (or becomes, by other rules) "tainted" with the dynamic type in the contravariant parts.

Definition 3.2 (Polarity of a type variable in a type)**.** *The set of positive variables and the set of negative variables in a type T are $\mathbb{P}^+(T)$ and $\mathbb{P}^-(T)$, respectively, defined as follows. Let s range over $\{+,-\}$; let $\overline{s}$ be $-$ when s is $+$, and $+$ when s is $-$.*

- $\mathbb{P}^+(X) = \{X\}, \mathbb{P}^-(X) = \{\}$
- $\mathbb{P}^s(\star) = \mathbb{P}^s(\bot) = \{\}$
- $\mathbb{P}^s(T_1 \to T_2) = \mathbb{P}^{\overline{s}}(T_1) \cup \mathbb{P}^s(T_2)$
- $\mathbb{P}^s(\{x_i^{\kappa_i} : T_i\}^{i\in[m]}) = \bigcup\{\mathbb{P}^{\overline{s}}(T_i) \mid i \in [m], - \in \kappa_i\} \cup \bigcup\{\mathbb{P}^s(T_i) \mid i \in [m], + \in \kappa_i\}$

For example, if the interface of the compiled program with existing code is $X_1 \to X_2$, then (F-COMP) reflects the following assumptions:

- We assume that there is a flow from $\star$ to X_1.
- We also assume that there is a flow from X_2 to $\star$, so that if (say) there is a local function that escapes through X_2, then (by other rules) we can derive a flow from $\star$ to the parameter type of the escaping function.

Similar considerations apply if the interface is an object type: if a writable property is annotated with a type variable, there is a flow from $\star$ to that type variable, and if a readable property is annotated with a type variable, there is a flow from that type variable to $\star$. (If a property is both, we assume both flows; if a property is neither, we assume neither.)

Other rules, namely (F-PULL), (F-FACTOR), and (F-TRAN), rely on a notion of kinding for type variables, defined below. Intuitively, a type variable has, for every type constructor that may form a solution for the type variable, a kind that encodes that solution.

Definition 3.3 (Kinds for a type variable)**.** *Kinds for a type variable X, ranged over by $\mathbb{K}^X$, are types of the form $\bot$, $X? \to X!$, $\{x_i^{\kappa_i} : X.x_i\}^{i\in[m]}$, or $\star$.*

Eventually, the solution of a type variable is derived by the kinds that flow into it, either directly or indirectly through other type variables. Thus, the rule (F-PULL) "pulls" any kind on the left of a type variable to any other type variable on its right.

The rule (F-FACTOR) factors a coercion from a type to a type variable into intermediate coercions through a corresponding kind for that type variable, computed as shown below. This factoring ensures that flows from types to a type variable are appropriately captured in solutions for that type variable.

Definition 3.4 (Kind of a type for a type variable)**.** *The kind $\|T\|_X$ of a type T, where T is not a type variable, for a type variable X is defined as follows:*

- $\|\bot\|_X = \bot$
- $\|\star\|_X = \star$
- $\|T_1 \to T_2\|_X = X? \to X!$
- $\|\{x_i^{\kappa_i} : T_i\}^{i\in[m]}\|_X = \{x_i^{\kappa_i} : X.x_i\}^{i\in[m]}$

The rule (F-TRAN) limits transitive flows through a type variable in two ways: it only considers a kind on the left, and it only considers a type on the right such that the kind on the left is *dynamically consistent* with the type on its right. Dynamic consistency $\preccurlyeq$ is a partial relation between types that are not type variables, defined as follows; it models coercions that never fail at run time.

Definition 3.5 (Dynamic Consistency)**.** *A type T is dynamically consistent with another type T' if T and T' are not type variables, and $T \preccurlyeq T'$ can be derived by any of the following rules.*

- $\bot \preccurlyeq T$
- $T \preccurlyeq \star, \star \preccurlyeq T$
- $T_1 \to T_2 \preccurlyeq T_1' \to T_2'$
- $\{x_i^{\kappa_i} : T_i\}^{i\in[m]} \preccurlyeq \{x_i^{\kappa_i'} : T_i'\}^{i\in[n]}$ *if $n \le m$, and for all $i \in [n]$, we have $\kappa_i' \subseteq \kappa_i$*

In combination with (F-FACTOR), the rule (F-TRAN) ensures that flows from types on the left of a type variable to types on the right of the type variable are taken into account, without computing a standard transitive closure. (In Section 4, we discuss why computing a standard transitive closure is undesirable.)

The remaining rules are fairly straightforward. (F-EXPFUNL), (F-EXPFUNR), (F-EXPOBJL), and (F-EXPOBJR) expand $\star$ on the left or right of a higher-order type to the appropriate shape. Finally, (F-SPLITFUN) and (F-SPLITOBJ) split flows between higher-order types into flows between their parts, respecting covariance and contravariance. Note that since we distinguish access capabilities for reading and writing object properties, the types of object properties are not necessarily invariant: they may be covariant (read-only), contravariant (write-only), invariant (read-write), or even abstract (no access).

3.3.2 Derivation of Solutions

Based on the flow judgment, we derive a solution map $\mathbb{I}$ that associates each type variable X to a type without type variables. We also extend $\mathbb{I}$ to a function $\widehat{\mathbb{I}}$ from types to types without type variables, such that $\widehat{\mathbb{I}}(T)$ is the type obtained by substituting each type variable X in type T by $\mathbb{I}(X)$.

To solve for X, let $\mathbb{T}^+(X)$ be the set of types T such that T is not a type variable, and $T \rhd X \checkmark$; we compute the least upper bound of the kinds of types for X in $\mathbb{T}^+(X)$, as defined below.

Definition 3.6 (Least upper bound of kinds)**.** *The least upper bound $\mathbb{K}_1^X \sqcup \mathbb{K}_2^X$ of two kinds $\mathbb{K}_1^X$ and $\mathbb{K}_2^X$ for X is defined as follows:*

- $\bot \sqcup \mathbb{K}^X = \mathbb{K}^X, \mathbb{K}^X \sqcup \bot = \mathbb{K}^X$
- $\star \sqcup \mathbb{K}^X = \star, \mathbb{K}^X \sqcup \star = \star$
- $(X? \to X!) \sqcup (X? \to X!) = X? \to X!$
- $\{x_i^{\kappa_i} : X.x_i\}^{i\in[m]} \sqcup \{y_j^{\kappa_j} : X.y_j\}^{j\in[n]} = \{z_k^{\kappa_k} : X.z_k\}^{k\in[p]}$, *where $\{z_k\}^{k\in[p]} = \{x_i\}^{i\in[m]} \cap \{y_j\}^{j\in[n]}$ and for all $i \in [m]$, $j \in [n]$, and $k \in [p]$, if $x_i = y_j = z_k$ then $\kappa_k = \kappa_i \cap \kappa_j$.*
- $(X? \to X!) \sqcup \{x_i^{\kappa_i} : X.x_i\}^{i\in[m]} = \star$
- $\{x_i^{\kappa_i} : X.x_i\}^{i\in[m]} \sqcup (X? \to X!) = \star$

The solution $\mathbb{I}(X)$ for X is then defined as $\widehat{\mathbb{I}}(\sqcup_{T\in\mathbb{T}^+(X)} \|T\|_X)$. Such solutions are always well-founded, since kinds do not have cyclic dependencies.

3.4 Blame Recovery

Standard gradual type systems enforce that the coercions in a compiled program satisfy *static consistency*, which is a partial relation between types that are not type variables, defined as follows:

Definition 3.7 (Static Consistency)**.** *A type T is statically consistent with another type T' if T and T' are not type variables, and $T \preceq T'$ can be derived by any of the following rules.*

- $\bot \preceq T$
- $T \preceq \star, \star \preceq T$
- $T_1 \to T_2 \preceq T'_1 \to T'_2$ *if* $T'_1 \preceq T_1$ *and* $T_2 \preceq T'_2$
- $\{x_i^{\kappa_i} : T_i\}^{i\in[m]} \preceq \{x_i^{\kappa'_i} : T'_i\}^{i\in[n]}$ *if* $n \leq m$*, and for all* $i \in [n]$*, we have* $\kappa'_i \subseteq \kappa_i$*, and if* $+ \in \kappa'_i$ *then* $T_i \preceq T'_i$*, if* $- \in \kappa'_i$ *then* $T'_i \preceq T_i$*.*

This static consistency relation extends similar relations defined separately for a gradually typed language of functions and for a gradually typed language of objects. We conjecture that programs in our language satisfy the blame theorem whenever they compile to programs whose coercions satisfy the static consistency relation, following similar type-safety results for existing languages.

However, the solution map $\mathbb{I}$ derived above does not guarantee static consistency. This means that blame guarantees offered by standard gradual type systems do not carry over with type inference. Fortunately, it is possible to "weaken" $\mathbb{I}$ to recover those blame guarantees, as follows. For any type variable X, let $\mathbb{T}^-(X)$ be the set of types T such that T is not a type variable and $X \rhd T \checkmark$. If there is any $T \in \mathbb{T}^-(X)$ such that $\mathbb{I}(X) \not\preceq T$, then $\mathbb{I}(X)$ is redefined to be $\star$. We can then prove the following theorem.

Theorem 3.8 (Blame recovery)**.** *Suppose that for every coercion* $T \rhd T'$ *in the compiled program, we have* $\widehat{*}(T) \preceq \widehat{*}(T')$*, where* $\widehat{*}$ *is the solution map that associates every type variable with* $\star$*. Then for every flow judgment* $T \rhd T' \checkmark$*, we have* $\widehat{\mathbb{I}}(T) \preceq \widehat{\mathbb{I}}(T')$*.*

3.5 Algorithmic Complexity

Although computation of closure may be performed by applying the flow rules in arbitrary order until fixpoint, an effective algorithm would apply them systematically, as described below; we can then reason about its efficiency.

Definition 3.9 (Flow computation)**.** *The computation of flow judgments proceeds in the following steps until fixpoint:*

(1) Initially, rules (F-Base) *and* (F-Comp) *are applied until saturation.*
(2) Next, rules (F-Pull) *and* (F-Factor) *are applied until saturation.*
(3) Next, rules (F-Tran)*,* (F-ExpFunL)*,* (F-ExpObjL)*,* (F-ExpFunR)*,* (F-ExpObjR) *are applied until saturation.*
(4) Next, rule (F-SplitFun) *and* (F-SplitObj) *is applied until saturation.*
(5) Finally, control returns to step (2), where if no new flow judgments are derived, the algorithm terminates.

We prove that our algorithm terminates and is efficient.

Theorem 3.10 (Termination)**.** *Flow computation terminates.*

Proof. Recall that all type variables in the system are distinct, so they have unique depths. Let d be the maximum depth of types in the system after step (1).

Steps (2) and (3) introduce types of depth $\leq d$ in the system. Step (4) introduces types of depth $\leq d-1$. Finally, in step (5), types of depth d and type variables of depth 1 can be thrown away. This means that maximum depth of types in the system after step (5) is $d-1$.

Thus, flow computation must terminate in $\leq d$ iterations. □

Theorem 3.11 (Time Complexity)**.** *The time complexity of flow computation is quadratic in the size of the program.*

Proof. Let k be the number of type variables, n be the number of types other than type variables, and d be the maximum depth of types after step (1).

Step (2) takes $O(k^2)$ time. Step (3) takes $O(kn)$ time, and increases the number of types by some factor w that denotes the maximum width of types. Step (4) takes $O(n^2)$ time, and increases the number of types by the same factor w.

Thus, before step (5), the total time taken is $O((k+n)^2)$, and after step (5), we have wk variables and wn types other than type variables.

Since the algorithm terminates in $\leq d$ iterations, the total time complexity is $O(w^{2d}(k+n)^2)$. Typically, w and d are small constants, so the time complexity is almost quadratic in the number of types $k+n$. In general, if the size of the program is N, then $w^d(k+n) = O(N)$. Thus the total time complexity is $O(N^2)$. □

3.6 Evaluation

We now describe the evaluation semantics of our language, which enables us to prove that our type inference algorithm is sound.

Let ℓ range over locations. A stack S is a sequence of locations. The syntax of values is as follows.

$$\begin{array}{lcll} \text{value} & v & ::= & \langle \bot \rhd \cdots \rhd T\rangle\ \mathsf{null} \\ & & | & \langle T_1 \to T_2 \rhd \cdots \rhd T\rangle\ \lambda^S x.\ t \\ & & | & \langle \{x_i^{\kappa_i} : T_i\}^{i\in[m]} \rhd \cdots \rhd T\rangle\ \ell \end{array}$$

A value of the form $\langle T \rhd T\rangle\ u$ is abbreviated as $u :: T$. Furthermore, we use the notation $\langle T' \rhd \cdots \rhd T''\rangle\ \langle T \rhd \cdots \rhd T'\rangle\ u$ to denote $\langle T \rhd \cdots \rhd T' \rhd \cdots \rhd T''\rangle\ u$. Unlike previous work that focuses on space-efficient implementations of gradually typed languages [19], our dynamic semantics admits unbounded chains of coercions to simplify the specification and proof of soundness: essentially, we keep coercions in symbolic form and check that they normalize "along the way."

A record is a map from variables to values. A heap H is a map from locations to records.

We rely on the following operations for querying/updating variables on the stack through the heap.

$$H[x \Mapsto v]_{S,\ell} = \begin{cases} H[\ell \mapsto H(\ell)[x \mapsto v]] & \text{if } x \in \mathtt{dom}(H(\ell)) \\ H[x \Mapsto v]_S & \text{if } x \notin \mathtt{dom}(H(\ell)) \end{cases}$$

$$H[x]_{S,\ell} = \begin{cases} H(\ell)(x) & \text{if } x \in \mathtt{dom}(H(\ell)) \\ H[x]_S & \text{if } x \notin \mathtt{dom}(H(\ell)) \end{cases}$$

The evaluation judgment is $S \vdash (H,t) \Downarrow_\sigma (H',v)$, where σ is a solution map, which associates type variables to types without type variables. Figure 5 lists the rules for deriving evaluation judgments.

The rules are fairly standard, except that the coercions on values are symbolic: the type variables in those coercions are not substituted by their solutions, but instead the solutions are looked up when normalizing the coercions. This is convenient for our soundness proof. Normalization is defined as follows.

Definition 3.12 (Normalization)**.** *The chain of coercions* $T_0 \rhd \cdots \rhd T_{n+1}$ *normalizes under* σ *if for all* $j \in [n+1]$*,* $\sigma(T_0) \preccurlyeq \sigma(T_j)$*.*

Furthermore, a value v *normalizes under* σ*, written* $v \Downarrow_\sigma$*, if it is of the form* $\langle T_0 \rhd \cdots \rhd T_{n+1}\rangle\ u$ *and* $\langle T_0 \rhd \cdots \rhd T_{n+1}\rangle$ *normalizes under* σ*.*

The deconstruction rules for functions and objects, namely (E-App), (E-PropR), and (E-PropW), rely on the following standard splitting rules for higher-order coercions.

$$\frac{\forall i \in [n+1].\quad T_i \multimap \{x_j^{\{+\}} : T'_i\}}{\langle T_0 \ldots T_{n+1} \rhd \{x_j^{\{+\}} : U_j\}\rangle \multimap\!\!\bullet \{x_j^{\{+\}} : \langle T'_0 \ldots T'_{n+1}\rangle\}}$$

$$\frac{\forall i \in [n+1].\quad T_i \multimap \{x_j^{\{-\}} : T'_i\}}{\langle T_0 \ldots T_{n+1} \rhd \{x_j^{\{-\}} : U_j\}\rangle \multimap\!\!\bullet \{x_j^{\{-\}} : \langle T'_{n+1} \ldots T'_0\rangle\}}$$

Evaluation judgment $S \vdash (H, t) \Downarrow_\sigma (H', v)$

(E-NULL)

$$\frac{}{S \vdash (H, \mathsf{null}) \Downarrow_\sigma (H, \mathsf{null} :: \bot)}$$

(E-FUN)

$$S \vdash (H, \mathsf{fun}\ (x : T_1)\ t_2 : T_2) \Downarrow_\sigma (H, \lambda^S x.\ t_2 :: T_1 \to T_2)$$

(E-OBJ)

$$\frac{\begin{array}{c}\ell \text{ is fresh} \qquad H_1 = H[\ell \mapsto [x_1 \mapsto \mathsf{null}, \dots, x_n \mapsto \mathsf{null}]] \\ \forall i \in [m].\quad S, \ell \vdash (H_i, t_i) \Downarrow_\sigma (H'_i, v_i) \quad H'_i[x_i \Mapsto v_i]_{S,\ell} = H_{i+1}\end{array}}{S \vdash (H, \{x_i^{\kappa_i} : T_i = t_i\}^{i\in[m]}) \Downarrow_\sigma (H_{m+1}, \ell :: \{x_i^{\kappa_i} : T_i\}^{i \in [m]})}$$

(E-PROPR)

$$\frac{\begin{array}{c}S \vdash (H, t) \Downarrow_\sigma (H', \langle \mathcal{C} \rangle\ \ell) \qquad \mathcal{C} \mathrel{-\!\!\bullet} \{x_j^{\{+\}} : \langle \mathcal{C}_j \rangle\} \\ H'(\ell) = R \qquad v_j = R(x_j) \qquad \langle \mathcal{C}_j \rangle\ v_j \ \Downarrow_\sigma\end{array}}{S \vdash (H, t.x_j) \Downarrow_\sigma (H', \langle \mathcal{C}_j \rangle\ v_j)}$$

(E-PROPW)

$$\frac{\begin{array}{c}S \vdash (H, t) \Downarrow_\sigma (H', \langle \mathcal{C} \rangle\ \ell) \qquad \mathcal{C} \mathrel{-\!\!\bullet} \{x_j^{\{-\}} : \langle \mathcal{C}_j \rangle\} \\ S \vdash (H', t_j) \Downarrow_\sigma (H_j, v_j) \qquad \langle \mathcal{C}_j \rangle\ v_j \ \Downarrow_\sigma \\ H_j(\ell) = R \qquad H'' = H_j[\ell \mapsto R[x_j \mapsto \langle \mathcal{C}_j \rangle\ v_j]]\end{array}}{S \vdash (H, t.x_j = t_j) \Downarrow_\sigma (H'', \langle \mathcal{C}_j \rangle\ v_j)}$$

(E-APP)

$$\frac{\begin{array}{c}S \vdash (H, t) \Downarrow_\sigma (H', \langle \mathcal{C} \rangle\ \lambda^{S'} x.t_2) \qquad \mathcal{C} \mathrel{-\!\!\bullet} \mathcal{C}_1 \to \mathcal{C}_2 \\ S \vdash (H', t_1) \Downarrow_\sigma (H_1, v_1) \qquad \langle \mathcal{C}_1 \rangle\ v_1 \ \Downarrow_\sigma \\ \ell \text{ is fresh} \qquad H'' = H_1[\ell \mapsto [x \mapsto \langle \mathcal{C}_1 \rangle\ v_1]] \\ S', \ell \vdash (H'', t_2) \Downarrow_\sigma (H_2, v_2) \qquad \langle \mathcal{C}_2 \rangle\ v_2 \ \Downarrow_\sigma\end{array}}{S \vdash (H, t\ (t_1)) \Downarrow_\sigma (H_2, \langle \mathcal{C}_2 \rangle\ v_2)}$$

(E-VARR)

$$\frac{H[x]_S = v}{S \vdash (H, x) \Downarrow_\sigma (H, v)}$$

(E-VARW)

$$\frac{S \vdash (H, t) \Downarrow_\sigma (H', v) \qquad H'[x \Mapsto v]_S = H''}{S \vdash (H, x = t) \Downarrow_\sigma (H'', v)}$$

(E-CAST)

$$\frac{S \vdash (H, t_1) \Downarrow_\sigma (H', v_1) \qquad \langle T_1 \rhd T_2 \rangle\ v_1 \ \Downarrow_\sigma}{S \vdash (H, \langle T_1 \rhd T_2 \rangle\ t_1) \Downarrow_\sigma (H', \langle T_1 \rhd T_2 \rangle\ v_1)}$$

(E-IF)

$$\frac{\begin{array}{c}S \vdash (H, t) \Downarrow_\sigma (H', v) \\ v \neq \mathsf{null} \Rightarrow i = 1 \qquad v = \mathsf{null} \Rightarrow i = 2 \\ S \vdash (H', t_i) \Downarrow_\sigma (H'', v')\end{array}}{S \vdash (H, \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \Downarrow_\sigma (H'', v')}$$

Figure 5. Evaluation

$$\frac{\forall i \in [n+1].\quad T_i \multimap T'_i \to T''_i}{\langle T_0 \dots T_{n+1} \rhd U_1 \to U_2 \rangle \mathrel{-\!\!\bullet} \langle T'_{n+1} \dots T'_0 \rangle \to \langle T''_0 \dots T''_{n+1} \rangle}$$

3.7 Soundness

Since we infer more precise types where there were less precise types ($\star$), the interesting direction for soundness is to establish, as above, that we do not introduce any run-time errors. The other direction is trivial, and can be established by reasoning about positive and negative blames [24] (see Section 6).

We prove the following soundness theorem for our type inference algorithm, which says that if a compiled program evaluates to a value with dynamic types for type variables, then it evaluates to the same value with inferred types for type variables. (Recall that the dynamic semantics is symbolic in type variables, but ensures that any coercions in values normalize.)

Theorem 3.13 (Soundness)**.** *Let $\varnothing \vdash t \hookrightarrow t' :: T$. Let $\mathbb{I}$ be the inferred solution map for $t' :: T$, and let $*$ be the solution map that associates every type variable with $\star$. If $\varnothing \vdash (\varnothing, t') \Downarrow_* (H', v)$, then $\varnothing \vdash (\varnothing, t') \Downarrow_{\mathbb{I}} (H', v)$.*

Soundness follows as a corollary of our main lemma, Term Correspondence (Lemma 3.16, see below), which states not only (i) there a correspondence between reductions in the original program and reductions in the program with inferred types, but also (ii) any coercions that may be generated at run time have already been generated at compile time (via closure computation). Of course, (ii) is a crucial invariant to show (i), since it means that the solutions we compute at compile time behave as expected at run time.

Definition 3.14 (Knowledge of coercions)**.** *The chain of coercions $T_0 \rhd \dots \rhd T_{n+1}$ is known if for all $j \in [n+1]$, $T_{j-1} \rhd T_j \checkmark$.*

Definition 3.15 (Compatibility of type environment with stack and heap)**.** *The type environment Γ is compatible with the stack S under heap H, written $\Gamma \sim_H S$, if whenever $\Gamma(x) = T_{n+1}$, we have $H[x]_S = \langle T_0 \rhd \dots \rhd T_{n+1} \rangle\ v$ and $\langle T_0 \rhd \dots \rhd T_{n+1} \rangle$ is known.*

Lemma 3.16 (Term Correspondence)**.** *Let $\Gamma \vdash t \hookrightarrow t' :: T$ and $\Gamma \sim_H S$. If $S \vdash (H, t') \Downarrow_* (H', \langle T_0 \rhd \dots \rhd T_{n+1} \rangle\ v)$ then:*

- *$S \vdash (H, t') \Downarrow_{\mathbb{I}} (H', \langle T_0 \rhd \dots \rhd T_{n+1} \rangle\ v$*
- *$\langle T_0 \rhd \dots \rhd T_{n+1} \rangle$ is known*

The proof of Term Correspondence requires Value Correspondence (Lemma 3.17, see below), and two other lemmas on knowledge of coercions, Function Flow and Object Flow, which are used in the cases of function application and object property access.

Lemma 3.17 (Value Correspondence)**.** *If $T_0 \rhd \dots \rhd T_{n+1}$ is known and normalizes under $*$, then it normalizes under $\mathbb{I}$.*

Lemma 3.18 (Function Flow)**.** *If $T_0 \to T'_0 \rhd T''_1 \rhd \dots \rhd T''_{n+1}$ is known and $T''_i \multimap T_i \to T'_i$ for all $i \in [n+1]$, then $T_n \dots T_1 \rhd T_0$ is known and $T'_0 \rhd T'_1 \dots T'_n$ is known.*

Lemma 3.19 (Object Flow)**.** *If $\{x_{i-1}^{\kappa_{i-1}} : T_{i-1}\}^{i \in [m]} \rhd T''_1 \rhd \dots \rhd T''_{n+1}$ is known and $T''_i \multimap \{x_0^{\kappa'_0} : T_i\}$ for all $i \in [n+1]$, then $T_n \dots T_1 \rhd T_0$ is known if $\kappa'_0 = \{-\}$ and $T'_0 \rhd T'_1 \dots T'_n$ is known if $\kappa'_0 = \{+\}$.*

The proofs of these lemmas in turn require the closure rules, and the following two basic lemmas on dynamic consistency.

Lemma 3.20 (Kind Ordering)**.** *If $\sqcup_{T \in \mathcal{T}} T = T'$ then for all $T \in \mathcal{T}$, we have $T \preccurlyeq T'$.*

Lemma 3.21 (Monotonicity)**.** *If $X \rhd Y \checkmark$ then $\mathbb{I}(X) \preccurlyeq \mathbb{I}(Y)$.*

Furthermore, we can prove the following soundness theorem that says that our type inference is compositional.

Theorem 3.22 (Compositional Soundness)**.** *Suppose that $\varnothing \vdash t_1 \hookrightarrow t'_1 :: T_1$. Let $\mathbb{I}$ be the inferred solution map for $t'_1 :: T_1$, and $*$ be the solution map that associates every type variable with $\star$. Let t be a program without type variables, such that $\varnothing \vdash t \hookrightarrow t' :: \widehat{*}(T_1) \to T_2$. Let $t''_1 = \langle T_1 \rhd \widehat{*}(T_1) \rangle\ t'_1$. Then $\varnothing \vdash (\varnothing, t'\ (t''_1)) \Downarrow_* (H, v_2)$ implies $\varnothing \vdash (\varnothing, t'\ (t''_1)) \Downarrow_{\mathbb{I}} (H, v_2)$.*

Proof. The new coercion is $T_1 \rhd \widehat{*}(T_1)$. However, by (F-COMP) we already know coercions of the form $X \rhd \star$ for positive type variables X and $\star \rhd X'$ for negative type variables X' in T_1. So by Term Correspondence, the composition is sound. $\square$

Finally, in Section 4.4 we discuss the adequacy of our inferred types, and conjecture that our algorithm is *relatively* complete (i.e., the solution maps it infers are optimal in a restricted family of possible solution maps). Unfortunately, we cannot compete with manual reasoning: we are working in a setting that admits reasoning outside the static type system via $\star$.

4. Discussion

4.1 Feature Extensions

Base Types Base types in our language can be modeled as kinds. Kinds correspond to type constructors, and the base types are type constructors of arity 0. By Definition 3.3, any base type is a kind for any X, and by Definition 3.4 the kind of a base type for a type variable as that base type itself. Finally, in Definition 3.6 the least upper bounds over base types takes into account partial subtyping relations between base types in the obvious manner.

Classes and Nominal Subtyping We do not model classes in our formal language, but could do so by closely following the treatment of objects. In particular, instance methods are modeled as read-only properties and instance fields are modeled as read-write properties; private properties are also modeled. Our algorithm and its guarantees carry over to a language with nominal subtyping, if the compiler ensures that nominal subtyping implies structural subtyping. For example, in ActionScript, nominal subtyping of objects is available via subclassing. The ActionScript compiler ensures that when class `B` is a subclass of class `A`, the structural type of `B`-objects is a subtype of the structural type of `A`-objects, by checking that the properties in `A` that are overridden in `B` are related by subtyping, taking into account read/write polarities.

4.2 Limited Transitivity

Previous type inference algorithms for object-oriented languages with subtyping [14, 15] involve taking full transitive closure of subtyping constraints, which makes them $O(n^3)$, where n is the size of the program. In contrast, we have designed our closure rules for flows so that transitivity is limited, which not only makes our algorithm $O(n^2)$, but also allows for more precise type inference. For example, consider this sequence of flows: `Number` $\rhd X \rhd Y \rhd$ `Boolean`. Our core algorithm infers $X =$ `Number` and $Y =$ `Number`. Now, if we want our solutions to satisfy the blame theorem, we fall back to $Y = \star$, since $Y =$ `Number` is inconsistent with $Y \rhd$ `Boolean`. However, we do not need to fall back to $X = \star$. In contrast, full transitivity would have lost the information that there is no immediate dynamic consistency check between X and `Boolean`.

4.3 Performance Optimizations

As we will see in our experiments, adding more precise types improves run-time performance in most cases. But more precise types can also *degrade* run-time performance by introducing new run-time conversions. It does not matter whether these precise types are added by type inference or manually—the effect is the same. For example, consider the following program:

```
function foo(x:*):Number { return x + 1 }
function bar(max:Number):Number {
    var sum:Number = 0;
    var y = 1;
    while(sum < max) { sum = sum + foo(y) }
    return sum
}
```

The function `foo` could be part of some external library which we cannot modify. Without any type annotation, variable `y` in `bar` will be assigned $\star$ type. So, the function call `foo(y)` will not involve any run-time conversions for `y`, since the parameter type for `foo` is also $\star$. But if we annotate `y` with `Number`, either using our type inference algorithm or manually, function call `foo(y)` will result in a run-time conversion from `Number` to $\star$ for `y`.

We observe that the fix for this problem lies in the runtime rather than our inference algorithm. Our type inference algorithm tries to infer more precise types for unannotated variables. The runtime should use these precise types to optimize the program. In particular, it can do type specialization [10] for `foo`: i.e., it can create a copy of `foo` which takes a `Number` argument, and patch the function call `foo(y)` in `bar` to this new copy. (Type specialization already happens in ActionScript for numeric operations.) The resulting program will be more optimized than the unannotated version.

4.4 Adequacy of Inferred Types

We now analyze how precise our inferred types are. Our closure algorithm ensures that if there exists some flow from a concrete type T to a type variable X via other type variables, we consider T when computing the solution for X. We do this to make sure that we do not introduce any new run-time errors by missing out some types that could flow into X. We promote the solution for X to $\star$ when two different kinds flow into X or $\star$ flows into X explicitly.

One could give more precise types than our algorithm by reasoning outside the type system. For example, consider the following program:

```
48 var x:X, y:Y;
49 if(b) { x = 1 } else { x = false };
50 if(b) { y = x } else { y = 1 }
```

We generate `Number` $\rhd X$ and `Boolean` $\rhd X$ on line 49, and $X \rhd Y$ and `Number` $\rhd Y$ on line 50. When we do closure, we add `Boolean` $\rhd Y$ also, and we infer $X = \star$ and $Y = \star$. Whereas, with path-sensitive reasoning (outside the type system), the programmer can argue that Y need not be $\star$, it can be `Number`.

Blame recovery may also reduce the precision of our solutions. For example, if the generated flows are `Number` $\rhd X$, $X \rhd Y$, and $Y \rhd$ `Boolean`, we infer $X =$ `Number` and $Y =$ `Number`. When we do blame recovery, we see that the coercion $Y \rhd$ `Boolean` is not satisfied, and so we promote Y to $\star$. Whereas, the programmer could get away by annotating Y as `Number` and making sure that $Y \rhd$ `Boolean` does not happen at run time, again by reasoning outside the type system.

However, we expect that if reasoning outside the type system is banned, then our algorithm indeed infers optimal solutions. Formally, we can define a subset of the static consistency relation that forces the use of purely static reasoning (standard subtyping) for inflows. We then conjecture that there are no other solutions that are naïve subtypes of the solutions we infer upon blame recovery, and that satisfy the above relation.

5. Experiments

We have implemented our type inference algorithm for ActionScript. Our implementation takes an existing ActionScript program as input, and returns the same program with inferred types added to the source as output. The output program can then be compiled and run using the compiler and VM, and compared with the input program.

5.1 Methodology

We use the SunSpider [20] and V8 [23] benchmarks to evaluate our type inference algorithm. These are standard benchmarks used to measure the performance of JavaScript implementations; we use the ActionScript version of these benchmarks, which are part of the test suite of the ActionScript VM.

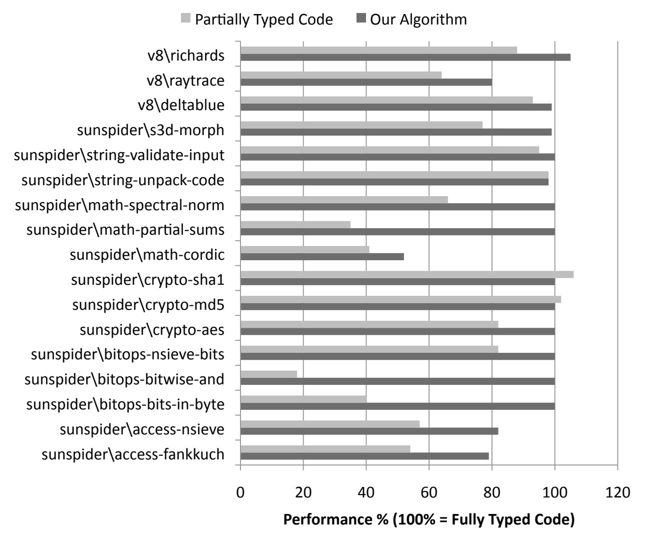

Figure 6. Comparison of our algorithm with partially typed code

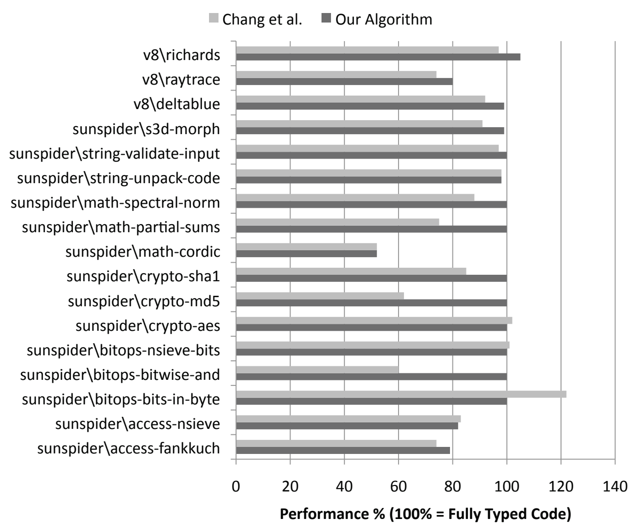

Figure 7. Comparison of our algorithm with Chang et al.

In their original form, these benchmarks are *fully typed*, i.e. their source code has complete type information. We remove all type annotations, except in those parts of the interface that are required by our algorithm to be explicitly annotated. We then run our algorithm on these *partially typed* benchmarks.

With performance of the fully typed benchmarks as the baseline, we compare the performance of partially typed benchmarks, with and without types added by our inference algorithm. We also compare the results of our algorithm on partially typed benchmarks with the results of Chang et al. [7], who have recently implemented a much simpler type inference algorithm (that only infers types for a subset of local variables) along with several other high-level optimizations for ActionScript at bytecode level, and report performance increases due to the added type information.

5.2 Results

5.2.1 Comparison with Partially Typed Code

Figure 6 compares the performance of partially typed benchmarks, with and without types added by our inference algorithm. Overall, our algorithm gives an average 1.6x performance improvement over partially typed benchmarks, with a maximum improvement of 5.6x.

We are able to recover $\sim$100% performance (i.e. the performance of fully typed benchmarks) in 13 out of the 17 benchmarks. For `v8\richards`, our performance is higher than the fully typed version. There are some private class variables in the benchmark which are typed as $\star$ in the fully typed version, whereas our algorithm is able to infer more precise types for them, resulting in an increased performance.

In the fully typed version of `sunspider\math-cordic`, an object is retrieved from an array and implicitly cast to an integer via an explicit `int` annotation. Since arrays are untyped in ActionScript, our inference algorithm infers the type of retrieved object as $\star$. This one annotation is the reason that we could reach only $\sim$50% performance as compared to the fully typed benchmark. Similar explicit type annotations for properties accessed from the type `Object`, which are also untyped in ActionScript, hurt our performance in `v8\raytrace`. For `sunspider\access-nsieve` and `sunspider\access-fankkuch`, we infer some variables as `Number`, whereas in the fully typed benchmarks, they are typed as `int`. In ActionScript, `int` represents 32 bit integers whereas `Number` has size 64 bits and can represent integers, unsigned integers, and floating-point numbers. At run time, operations involving `int` are much faster than those involving `Number`. Since we do not implement a range analysis, we conservatively infer all numeric types to be `Number`. This hurts the performance in the above benchmarks. Our algorithm can be augmented with a simple range analysis to mitigate this problem.

5.2.2 Comparison with Chang et al. [7]

Figure 7 compares the performance of partially typed benchmarks, with types added by our inference algorithm and with types added by inference algorithm of Chang et al. In 11 out of the 17 benchmarks, our numbers are better than them. In 5 out of the 17, the numbers are almost equal. For `bitops-bits-in-byte`, they report that the higher performance is because of a different intermediate code representation they implement in the ActionScript JIT compiler. This effect is independent of the type inference algorithm.

5.2.3 Performance Degradation after Type Inference

In Figure 6, we see that for `crypto-sha1`, the performance degrades after adding precise type information to the partially typed code. This general issue was also found by [7] (where the effect is even worse), and is already discussed in Section 4.3.

6. Related Work

Gradual Type Inference The only previous work on gradual type inference is the unification-based technique of [18], which is not suitable for an object-oriented language with subtyping.

Soft Typing Overall, our goal is the same as that of soft typing [6]: to improve run-time performance by eliminating run-time type checks where ever possible. However, to our knowledge, all the soft type inference systems proposed to date [2, 25] are based on unification, which is unsuitable for object-oriented languages with subtyping. Since we do not aim to infer polymorphic types in ActionScript, we are interested in algorithms with polynomial complexity. Furthermore, treating uses and definitions asymmetrically enables us to infer more precise types than soft type inference. Finally, existing soft type inference systems have not considered problems of soundness in the presence of partial compilation, whereas our algorithm retrofits type inference on existing programs under the assumption that complete source code may not be available for analysis. This restriction of preserving semantics of code outside the compilation unit implies that we must infer types for only those parts of the program that we can see all the writes to, i.e., those parts of the program that do not "escape" the compilation unit.

Blame Our algorithm seems to share deep connections with the blame calculus [19, 24] and other coercion calculi [11]; exploring these connections should be interesting future work. In particular, the blame calculus defines three new subtyping relations: $<:^+$ (positive subtyping), $<:^-$ (negative subtyping), and $<:_n$ (naïve subtyping), such that $S <:_n T$ denotes that S is more precise than T, and holds if and only if $S <:^+ T$ and $T <:^- S$. In particular, we have $S <:^+ \star$ and $\star <:^- T$. The main result in [24] is that if $S <:^+ T$ then a run-time cast error from S to T cannot be blamed on the term to which the cast is attached, and if $S <:^- T$ then a run-time cast error from S to T cannot be blamed on the context in which the cast appears.

The solutions of our algorithm are related to the default types by $<:_n$, so we can say that our solutions are more precise than the default types. In a context that previously expected the default type, we now effectively introduce a cast from a more precise type; this means that any blame for a run-time cast error must lie with the context, so there is nothing to prove about the program. On the other hand, wherever we infer a more precise type for a context, we effectively introduce a cast from the default type; this means that any blame for a run-time cast error must lie with the program, i.e., we must prove that a program that executed successfully before type inference continues to execute successfully after type inference—as we do in this paper.

Combining Static Typing and Dynamic Typing There has been a lot of interest in exploring ways to mix typed and untyped code, e.g., via occurrence typing [22], gradual typing [16, 17], hybrid typing [8], and like typing [26]. In these systems, types are supplied by the user. In contrast, our work focuses on type inference, which is complementary.

Static Type Inference Static type inference has been explored for dynamic languages, including Self [1], Ruby [9], Python [4], JavaScript [5, 21], and Scheme [25]. There is also a long history of work on type inference for languages with subtyping [14, 15].

Dynamic Type Inference Our type system gradually infers some static types, but is still bound by the limitations of static analysis on programs that use dynamic types. As such, we believe that dynamic type inference would be useful to improve the precision of those dynamic types which we cannot eliminate in a program. Recent work has explored dynamic techniques for type inference and type specialization [3, 10] for dynamic languages. As future work, we plan to explore such combinations in the just-in-time (JIT) compiler underlying the ActionScript VM.

7. Conclusion

In this paper, we design a type inference algorithm that can improve the performance of existing gradually typed programs, without introducing any new run-time failures. The distinguishing features of the algorithm lie in its asymmetric treatment of inflows and outflows, and its encoding of an escape analysis to preserve backward-compatibility. We prove that our algorithm is sound and efficient, and demonstrate its applicability on a mainstream gradually typed language, ActionScript.

Acknowledgments We would like to thank Bernd Mathiske, Dimitrios Vytiniotis, and several anonymous reviewers for their helpful comments and suggestions. This work was done while the first author was an intern at Advanced Technology Labs, Adobe Systems.

References

[1] O. Agesen, J. Palsberg, and M.I. Schwartzbach. Type Inference of SELF. *ECOOP*, 1993.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *POPL*, pages 163–173, 1994.

[3] J. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic Inference of Static Types for Ruby. In *POPL*, pages 459–472. ACM, 2011.

[4] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*. ACM, 2007.

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, 2005.

[6] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[7] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, and M. Franz. The Impact of Optional Type Information on JIT Compilation of Dynamically Typed Languages. In *DLS*. ACM, 2011.

[8] C. Flanagan. Hybrid Type Checking. In *POPL*, pages 245–256. ACM, 2006.

[9] M. Furr, J. D. An, J. S. Foster, and M. Hicks. Profile-Guided Static Typing for Dynamic Scripting Languages. In *OOPSLA*, 2009.

[10] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478. ACM, 2009.

[11] F. Henglein. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming*, 22(3):197 – 230, 1994.

[12] D. Herman, A. Tomb, and C. Flanagan. Space-Efficient Gradual Typing. *Trends in Functional Programming*, 2007.

[13] C. Moock. *Essential ActionScript 3.0*. O'Reilly, 2007.

[14] J. Palsberg. Efficient Inference of Object Types. In *LICS*, pages 186–195. IEEE, 1994.

[15] F. Pottier. A Framework for Type Inference with Subtyping. In *ICFP*, pages 228–238. ACM, 1998.

[16] J. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP*, pages 2–27. Springer-Verlag, 2007.

[17] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006.

[18] J. G. Siek and M. Vachharajani. Gradual Typing with Unification-Based Inference. In *DLS*, pages 1–12. ACM, 2008.

[19] J. G. Siek and P. Wadler. Threesomes, With and Without blame. In *POPL*, pages 365–376. ACM, 2010.

[20] SunSpider Benchmarks, 2010. `http://www.webkit.org/perf/sunspider/sunspider.html`.

[21] P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, 2005.

[22] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, 2008.

[23] V8 Benchmarks, 2011. `http://code.google.com/apis/v8/benchmarks.html`.

[24] P. Wadler and R. B. Findler. Well-Typed Programs Can't Be Blamed. In *ESOP*, pages 1–16. Springer-Verlag, 2009.

[25] A. K. Wright and R. Cartwright. A Practical Soft Type System for Scheme. *ACM TOPLAS*, 19(1), 1997.

[26] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*, pages 377–388. ACM, 2010.