Gradual Typing with Unification-based Inference

Jeremy G. Siek
University of Colorado at Boulder
siek@colorado.edu

Manish Vachharajani
University of Colorado at Boulder
manishv@colorado.edu

ABSTRACT

Static and dynamic type systems have well-known strengths and weaknesses. *Gradual typing* provides the benefits of both in a single language by giving the programmer control over which portions of the program are statically checked based on the presence or absence of type annotations. This paper studies the combination of gradual typing and unification-based type inference with the goal of developing a system that helps programmers increase the amount of static checking in their program. The key question in combining gradual typing and type inference is how should the dynamic type of a gradual system interact with the type variables of a type inference system. This paper explores the design space and shows why three straightforward approaches fail to meet our design goals. This paper presents a new type system based on the idea that a solution for a type variable should be as informative as any type that constrains the variable. The paper also develops an efficient inference algorithm and proves it sound and complete with respect to the type system.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and FeaturesF.3.3 [**Logics and Program Constructs**]: Studies of Program Constructs—*Type structure*

General Terms

Languages, Algorithms

Keywords

dynamic typing, static typing, type inference, unification, gradual typing, simply typed lambda calculus

1. INTRODUCTION

Static and dynamic typing have complementary strengths, making them better for different tasks and stages of development. Static typing, used in languages such as Standard ML [31], provides full-coverage type error detection, facilitates efficient execution, and provides machine-checked documentation that is particularly helpful for maintaining consistency when programming in the large. The main drawback of static typing is that the whole program must be well-typed before the program can be run. Typing decisions must be made for all elements of the program, even for ones that have yet to stabilize, and changes in these elements can ripple throughout the program.

In a dynamically typed language, no compile-time checking is performed. Programmers need not worry about types while the overall structure of the program is still in flux, making dynamic languages suitable for rapid prototyping. Dynamic languages such as Perl, Ruby, Python, and JavaScript are popular for scripting and web applications where rapid prototyping is needed. The problem with dynamic languages is that they forgo the benefits of static typing: there is no machine checked documentation, execution is often less efficient, and errors are caught only at run-time, sometimes after deployment.

Gradual typing, recently introduced by Siek and Taha [47], enables programmers to mix static and dynamic type checking in a program by providing a convenient way to control which parts of a program are statically checked. The defining properties of a gradually typed language are:

1. Programmers may omit type annotations and run the program; run-time type checks preserve type safety.
2. Programmers may add type annotations to increase static checking. When all variables are annotated, all type errors are caught during compilation.
3. Statically typed regions of a program cannot be blamed for run-time type errors.

A number of researchers have further studied gradual typing over the last two years. Herman, Tomb, and Flanagan [21] developed space-efficient run-time support for gradual typing. Siek and Taha [48] integrated gradual typing with objects and subtyping. Wadler and Findler showed how to perform blame tracking and proved that the well-typed portions of a program can't be blamed [55]. Herman and Flanagan are adding gradual typing to the next version of JavaScript [20].

An important question, from both a theoretical and practical perspective, has yet to be answered: is gradual typing compatible with type inference? Type inference is common in modern functional languages and is becoming more common in mainstream languages [56, 19]. There are many flavors of type inference: Hindley-Milner inference [30], dataflow-based inference [13], Soft Typing [6], and local inference [38] to name a few. In this paper we study type inference based on unification [43], the foundation of Hindley-Milner inference and the related family of algorithms used in many functional languages [31, 36, 27].

The contributions of this paper are:

- An exploration of the design space that shows why three straightforward approaches to specifying the type system do not satisfy our design goals (Section 3). The three approaches are: 1) treat dynamic types as type variables, 2) check whether the program is well-typed after substitution, and 3) ignore dynamic types during unification.
- A new type system based on the idea that the solution for a type variable should be as informative as any type that constrains the variable (Section 4). We formalize this idea in a type system (Section 4.2) and prove that it satisfies the criteria for a gradually typed language (Section 4.3). The machine checked proofs are available in a companion technical report [46].
- An inference algorithm for the new type system (Section 5). We prove that the algorithm is sound and complete with respect to the type system and that the algorithm has almost linear time complexity (Section 5.3). The algorithm does not infer types that introduce unnecessary cast errors. The implementation is freely available at `http://ece.colorado.edu/~siek/gtubi.tar.gz`.

Before the main technical developments, we review gradual typing as well as traditional unification-based inference (Section 2). After the technical developments, we place our work in relation to the relevant literature (Section 6) and conclude (Section 7).

2. BACKGROUND

We review gradual typing in the absence of type inference, showing examples in a hypothetical variant of Python [53] that supports gradual typing but not type inference. We then review type inference in the absence of gradual typing.

2.1 Review of Gradual Typing

The incr function listed below has a parameter x and returns x + 1. The parameter x does not have a type annotation so the gradual type system delays checks concerning x inside the incr function until run-time, just as a dynamically typed language would.

```
def incr(x):
    return x + 1
a:int = 1
incr(a)
```

More precisely, because the parameter x is not annotated the gradual type system gives it the *dynamic type*, written ? for short. Next, consider how x is used inside of incr. To simplify the present discussion, suppose the + operator expects arguments of type int. The gradual type system allows the implicit coercion from type ? to int even though this kind of coercion could fail (like a down cast) and therefore must be dynamically checked.

To facilitate migrating code from dynamic to static checking, gradual typing allows for a mixture of the two. In the example above, we define a variable a of type int and invoke the dynamically typed incr function. Here the gradual type system allows an implicit coercion from int to ?. This is a safe coercion—it can never fail at run-time—however the run-time system needs to remember the type of the value so that it can check the type when it casts back to int inside of incr.

Gradual typing also allows implicit coercions among more complicated types, such as function types. In the following example, the map function has a parameter f annotated with the function type $\mathsf{int} \to \mathsf{int}$ and a parameter l of type int list.

```
def map (f:int→int, l:int list):
    return [f(x) for x in l]
def incr(x):
    return x + 1
a:int = 1
map(incr, [1, 2, 3]) # OK
map(a, [1, 2, 3]) # compile time type error
```

The function call map(incr, [1, 2, 3]) is allowed by the gradual type system, even though the type of the argument incr ($? \to \mathsf{int}$) differs from the type of the parameter ($\mathsf{int} \to \mathsf{int}$). The type system compares the two types structurally and allows the two types to differ in places where one of the types has a ?. Thus, the function call is allowed because the return types are equal and there is a ? in one of the parameter types. In contrast, map(a, [1, 2, 3]) elicits a compile-time error because argument a has type int whereas f is annotated with a function type.

Motivation for Implicit Coercions.

The goal of gradual typing is to enable a smooth migration between dynamic and statically typed code. In the gradual type system, a programmer adds type annotations to function parameters to migrate from dynamic to static, and the compiler inserts or removes run-time checks as needed. If instead the run-time checks were inserted by the programmer in the form of explicit casts, then every change to a parameter's type annotation would require the programmer to manually insert or remove casts everywhere the parameter is used. This extra work creates an unnecessary barrier to migrating code between dynamic and static typing.

Why Subtyping Does Not Work.

Gradual typing allows an implicit up-cast from any type to ?, similar to object-oriented type systems where Object is the top of the subtype lattice. However, gradual typing differs in that it also allows implicit down casts. This is *the* distinguishing feature of gradual typing and is what gives it the flavor of dynamic typing. Previous attempts at mixing static and dynamic typing, such as the Quasi-static Typing [50], tried to use subtyping but had to deal with the following problem. If the dynamic type is treated as both the top and the bottom of the subtype lattice (allowing both implicit up-casts and down-casts), then the lattice collapses to one point because subtyping is transitive. In other words, every type is a subtype of every other type and the type system no longer rejects any program, even ones with obvious type errors.

Consider the following program.

```
def add1(x : int) →int:
    return x + 1
add1(true)
```

Using true as an argument to the function add1 is an obvious type error but we have $\mathsf{bool} <: ?$ and $? <: \mathsf{int}$, so $\mathsf{bool} <: \mathsf{int}$. Thus the subtype-based type system would accept this program. Thatte partially addressed this problem by adding a post-pass after the type checker but this still did not result in a system that catches all type errors within fully annotated code [33].

The Consistency Relation.

Instead of using subtyping, the gradual type system uses a relation called *consistency* [47], written $\sim$. The intuition behind consistency is to check whether two types are equal in the parts where both types are defined (i.e. not ?). Here are a few examples:

$$\mathsf{int} \sim \mathsf{int} \quad \mathsf{bool} \not\sim \mathsf{int} \quad ? \sim \mathsf{int} \quad \mathsf{int} \sim ?$$
$$? \sim \mathsf{int} \to \mathsf{bool} \quad \mathsf{int} \to ? \sim ? \to \mathsf{int} \quad \mathsf{int} \to ? \sim \mathsf{int} \to \mathsf{bool}$$
$$\mathsf{int} \to ? \not\sim \mathsf{bool} \to ? \quad \mathsf{int} \to \mathsf{int} \not\sim \mathsf{int} \to \mathsf{bool}$$

The following is the inductive definition of the consistency relation. Here we limit the definition to function types but it can be extended to other type constructors such as object types [48]. We use the metavariable τ to range over arbitrary types and γ to range over ground types such as int and bool.

Type Consistency

$$\frac{}{\gamma \sim \gamma} \qquad \frac{}{\tau \sim ?} \qquad \frac{}{? \sim \tau} \qquad \frac{\tau_1 \sim \tau_3 \quad \tau_2 \sim \tau_4}{\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4}$$

The consistency relation is reflexive and symmetric but not transitive. The consistency relation is symmetric because we want to allow implicit coercions both to and from ? as explained above. The lack of transitivity is necessary for the gradual type system to still have a static flavor and reject some programs. For example, because $\mathsf{bool} \not\sim \mathsf{int}$, the gradual type system properly rejects the function call $\mathsf{add1(true)}$ in the above example.

At first glance, the consistency rule for function types may seem strange because it is not covariant and contravariant in the same way as a subtype relation. Because consistency is symmetric, it does not make sense to talk of covariance and contravariance: flipping $\tau_1 \sim \tau_3$ to $\tau_3 \sim \tau_1$ does not make a difference.

The syntax of the gradually typed lambda calculus ($\lambda^?_\to$) is shown below and the type system is reproduced in Figure 1. The gradual type system uses type consistency where a simple type system uses type equality. For example, the (APP2) rule in the gradually typed lambda calculus requires that the argument type τ_2 be consistent with the parameter type τ_1.

Syntax for $\lambda^?_\to$

Variables	x, y	$\in \mathbb{X}$	
Ground Types	γ	$\in \mathbb{G}$	$\supseteq \{\mathsf{bool}, \mathsf{int}, \mathsf{unit}\}$
Constants	c	$\in \mathbb{C}$	$\supseteq \{\mathsf{true}, \mathsf{false}, \mathsf{succ}, 0, (), \mathsf{fix}[\tau]\}$
Types	τ	$::=$	$? \mid \gamma \mid \tau \to \tau$
Expressions	e	$::=$	$x \mid c \mid e\ e \mid \lambda x{:}\tau.\ e$

$$\lambda x.\ e \equiv \lambda x{:}?.\ e$$
$$\mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \equiv (\lambda x{:}\ \tau.\ e_2)\ e_1$$

$$\boxed{\Gamma \vdash_g e : \tau}$$

(VAR) $$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash_g x : \tau_1}$$

(CNST) $$\Gamma \vdash_g c : \mathit{typeof}(c)$$

(APP1) $$\frac{\Gamma \vdash_g e_1 : ? \quad \Gamma \vdash_g e_2 : \tau}{\Gamma \vdash_g e_1\ e_2 : ?}$$

(APP2) $$\frac{\Gamma \vdash_g e_1 : \tau_1 \to \tau_3 \quad \Gamma \vdash_g e_2 : \tau_2 \quad \tau_1 \sim \tau_2}{\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

(ABS) $$\frac{\Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{\Gamma \vdash_g \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

Figure 1: The type system for $\lambda^?_\to$.

The dynamic semantics of the gradually typed lambda calculus is defined by a translation to an intermediate language with explicit casts and by an operational semantics for the intermediate language [47]. The translation to the intermediate language infers where casts, i.e. where the run-time type checks, are needed.

The gradually typed lambda calculus meets the three criteria for a gradually typed language discussed in Section 1.

1. For programs without type annotations, i.e., every variable is assigned the ? type, little static checking is performed because the consistency relation allows implicit coercions both to and from the ? type.

2. When there are no ?s in the program (either explicitly or implicitly), the type system is equivalent to a fully static type system because the consistency relation collapses to equality when there are no ?s, i.e., for any σ and τ that contain no ?s, $\sigma \sim \tau$ iff $\sigma = \tau$.

3. We define a *safe* region of a gradually typed program to be a region that only contains safe implicit coercions, that is coercions where the source type is a subtype of the target type. Implicit coercions are turned into explicit casts in the intermediate language and Wadler and Findler [55] show that, for a richer intermediate language, safe casts cannot be blamed for run-time type errors.

2.2 Review of Unification-based Type Inference

Type inference allows programmers to omit type annotations but still enjoy the benefits of static type checking. For example, the following is a well-typed Objective Caml program. The inference algorithm deduces that the type of function f is $\mathsf{int} \to \mathsf{int}$.

```
# let f x = x + 1;;
val f : int →int = ⟨fun⟩ (∗ Output of inference ∗)
```

The type inference problem is formulated by attaching a type variable, an *unknown*, to each location in the program. The job of the inference algorithm is to deduce a solution for these variables that obeys the rules of the type system. So, for example, the following is the above program annotated with type variables.

$$\mathsf{let}\ \mathsf{f}_\alpha\ \mathsf{x}_\beta = (\mathsf{x}_\gamma +_\delta 1_\chi)_\rho$$

The inference algorithm models the rules of a type system as equations that must hold between the type variables. For example, the type β of the parameter x must be equal to the type γ of the occurrence of x in the body of f. The parameter types of + (both are int) must be equal to the argument types γ and χ, and the return type of +, also int, must be equal to ρ. Ultimately, the type α of f must be equal to the function type $\beta \to \rho$ formed from the parameter type β and the return type ρ. This set of equations can be solved by unification [43]. A *substitution* is a mapping from type variables to types and can be extended to map types to types. The unification algorithm computes a substitution S such that for each equation $\tau_1 = \tau_2$, we have $S(\tau_1) = S(\tau_2)$.

A natural setting in which to formalize type inference is the simply typed lambda calculus with type variables ($\lambda^\alpha_\to$). The syntax is similar to $\lambda^?_\to$, but with type variables and no dynamic type. The type system for the simply typed lambda calculus is reproduced in Figure 2. The extension of this type system to handle type variables, given below, is also standard [37].

DEFINITION 1. *A term e of $\lambda^\alpha_\to$ is **well-typed** in environment Γ if there is a substitution S and a type τ such that $S(\Gamma) \vdash S(e) : \tau$.*

We refer to this approach to defining well-typedness for programs with type variables as *well-typed after substitution*.

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash x : \tau_1} \qquad \boxed{\Gamma \vdash e : \tau}$$

$$\Gamma \vdash c : \mathit{typeof}(c)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma(x \mapsto \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\ e : \tau_1 \to \tau_2}$$

Figure 2: The type system of the simply typed λ-calculus.

An inference algorithm for $\lambda^{\alpha}_{\to}$ can be expressed as a two-step process [57, 37, 16] that generates a set of constraints (type equalities) from the program and then solves the set of equalities with unification. Constraint generation for $\lambda^{\alpha}_{\to}$ is defined in Figure 3. The soundness and completeness of the inference algorithm with respect to the type system has been proved in the literature [57, 37].

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \mid \{\}} \qquad \boxed{\Gamma \vdash e : \tau \mid C}$$

$$\Gamma \vdash c : \mathit{typeof}(c) \mid \{\}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \mid C_1 \\ \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad (\beta \text{ fresh})\end{array}}{\Gamma \vdash e_1\ e_2 : \beta \mid \{\tau_1 = \tau_2 \to \beta\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma(x \mapsto \tau) \vdash e : \rho \mid C}{\Gamma \vdash \lambda x : \tau .\ e : \tau \to \rho \mid C}$$

Figure 3: The definition of constraint generation for $\lambda^{\alpha}_{\to}$.

In Section 4 we combine inference with gradual typing and in that setting we need to treat type variables with special care. If we follow the well-typed-after-substitution approach, type variables are substituted away before the type system is consulted. As an intermediate step towards integration with gradual typing, we give an equivalent definition of well-typed terms for $\lambda^{\alpha}_{\to}$ that combines the substitution S with the type system. The type system is shown in Figure 4 and the judgment has the form $S; \Gamma \vdash e : \tau$ which reads: e is well-typed because S and τ are a solution for e in Γ.

Formally, we use the following representation for substitutions, which is common in mechanized formalizations [32].

DEFINITION 2. *A **substitution** is a total function from type variables to types and its **dom** consists of the variables that are not mapped to themselves. Substitutions extend naturally to types, typing environments, and expressions. The $\circ$ operator is the functional composition of two substitutions.*

$$(\text{SVAR}) \quad \frac{\Gamma(x) = \tau}{S; \Gamma \vdash x : \tau} \qquad \boxed{S; \Gamma \vdash e : \tau}$$

$$(\text{SCNST}) \quad S; \Gamma \vdash c : \mathit{typeof}(c)$$

$$(\text{SAPP}) \quad \frac{\begin{array}{c}S; \Gamma \vdash e_1 : \tau_1 \quad S; \Gamma \vdash e_2 : \tau_2 \\ S(\tau_1) = S(\tau_2 \to \tau_3)\end{array}}{S; \Gamma \vdash e_1\ e_2 : \tau_3}$$

$$(\text{SABS}) \quad \frac{S; \Gamma(x \mapsto \tau_1) \vdash e : \tau_2}{S; \Gamma \vdash \lambda x : \tau_1 .\ e : \tau_1 \to \tau_2}$$

Figure 4: The type system for $\lambda^{\alpha}_{\to}$.

Theorem 1 states that the type system of Figure 4 is equivalent to Definition 1 and relies on the following two lemmas. The function FTV returns the free type variables within a type, type environment, or expression.

LEMMA 1. *If $S(\Gamma) \vdash S(e) : \tau$ and S is idempotent then $S(\tau) = \tau$.*

PROOF. Observe that if $S(\Gamma) \vdash S(e) : \tau$ then $\mathrm{FTV}(\tau) \cap \mathrm{dom}(S) = \emptyset$. Furthermore, if S is idempotent then $\mathrm{FTV}(\tau) \cap \mathrm{dom}(S) = \emptyset$ implies $S(\tau) = \tau$. $\square$

LEMMA 2. *If S idempotent and $S(\tau) = \tau_1 \to \tau_2$ then $S(\tau_2) = \tau_2$.*

PROOF. We have $\tau_1 \to \tau_2 = S(\tau) = S(S(\tau)) = S(\tau_1 \to \tau_2) = S(\tau_1) \to S(\tau_2)$. Thus $\tau_2 = S(\tau_2)$. $\square$

THEOREM 1. *The two type systems for $\lambda^{\alpha}_{\to}$ are equivalent.*

1. *Suppose S is idempotent. If $S(\Gamma) \vdash S(e) : \tau$, then there is a τ' such that $S; \Gamma \vdash e : \tau'$ and $S(\tau') = \tau$.*
2. *If $S; \Gamma \vdash e : \tau$, then $S(\Gamma) \vdash S(e) : S(\tau)$.*

PROOF. 1. $S(\Gamma) \vdash S(e) : \tau \implies S(\Gamma) \vdash S(e) : S(\tau)$ by Lemma 1. We prove by induction that $S(\Gamma) \vdash S(e) : S(\tau)$ implies there is a τ' such that $S; \Gamma \vdash e : \tau'$ and $S(\tau') = S(\tau)$. We use Lemma 1 in the (APP) case and Lemma 2 in the (ABS) case. Then using Lemma 1 once more gives us $S(\tau') = \tau$.
2. The proof is a straightforward induction on $S; \Gamma \vdash e : \tau$. $\square$

3. EXPLORATION OF THE DESIGN SPACE

We investigate three straightforward approaches to integrate gradual typing and type inference. In each case we give examples of programs that should be well-typed but are rejected by the approach, or that should be ill-typed but are accepted.

Dynamic Types as Type Variables.

A simple approach is to replace every occurrence of ? in the program with a fresh type variable and then do constraint generation and unification as presented in Section 2. The resulting system is fully static, not gradual. Consider the following program.

```
let z = ...
let f (x : int) = ...
let g (y : bool) = ...
let h (a : ?) = if z then f a else g a
```

Variable a has type ? and so a fresh type variable α would be introduced for its type. The inference algorithm would deduce from the function applications f a and g a that $\alpha =$ int and $\alpha =$ bool respectively. There is no solution to these equations, so the program would be rejected with a static type error. However, the program would run without error in a dynamically typed language given an appropriate value of z and input for h. Furthermore, this program type checks in the gradual type system of Figure 1 so it ought to remain valid in the presence of type inference.

The next example exhibits a different problem: the inference algorithm may not find concrete solutions for some variables and therefore indicate polymorphism in cases where there shouldn't be.

```
let f (x : int) (g : ? →?) =
    g x
```

Generating fresh type variables for the ?s gives us $\mathsf{g} : \alpha \to \beta$. Let γ be the type variable for the return type of f and the type of the expression g x. The only equation constraining γ is $\gamma = \beta$, so the return type of f is inferred to be β giving the impression that f is polymorphic. But if f is really polymorphic in β it should behave uniformly for any choice β [41, 54]. Suppose g is the identity function. Then f raises a cast error if $\beta = \mathsf{bool}$ but not if $\beta = \mathsf{int}$.

While treating occurrences of the dynamic type literally as type variables does not work, it is a promising direction. The solution we propose in Section 4 can be viewed as a more sophisticated variation on this theme.

Ignore Dynamic Types During Unification.

Yet another straightforward approach is to adapt unification by simply ignoring any unification of the dynamic type with any other type. However, this results in programs with even more unsolved variables than in the approach described above. Consider again the following program.

```
let f (x : int) (g : ? →?) =
    g x
```

From the function application, the inference algorithm would deduce $? \to ? = \mathsf{int} \to \beta$, where β is a fresh variable representing the result type of the application g x. This equality would decompose to $? = \mathsf{int}$ and $? = \beta$. However, if the unification algorithm does not do anything with $? = \beta$, we end up with β as an unsolved variable, giving the impression that f is parametric in β, just as above.

Well-typed After Substitution.

In Section 2 we presented the standard type system for $\lambda^{\alpha}_{\to}$, saying that a program is well typed if there is some substitution that makes the program well typed in $\lambda_{\to}$. We could do something similar for gradual typing, saying that a gradually typed program with variables is well typed if there exists a substitution that makes it well typed in $\lambda^{?}_{\to}$ (Figure 1).

It turns out that this approach is too lenient. Recall that to satisfy criteria 2 of gradual typing, for fully annotated programs the gradual type system should act like a static type system. Consider the following program that would not type check in a static type system because α cannot be both an int and a function type.

```
let f (g:α) = g 1
f 1
```

Applying the substitution $\{\alpha \mapsto ?\}$ produces a program that is well-typed in $\lambda^{?}_{\to}$.

The next example shows a less severe problem, although it still undermines the purpose of type inference, which is to help programmers increase the amount of static typing in their programs.

```
let x:α = x + 1
```

Again, the substitution $\{\alpha \mapsto ?\}$ is allowed, but it does not help the programmer. Instead, one would like to find out that $\alpha = \mathsf{int}$. In general, we need to be more careful about where ? is allowed as the solution for a type variable.

However, we cannot altogether disallow the use of ? in solutions because we want to avoid introducing run-time cast errors. Consider the program

```
let f (x:?) =
    let y:α = x in y
```

Here, the *only* appropriate solution for α is the dynamic type. Any other choice introduces an implicit cast to that type, which causes a run-time cast error if the function is applied to a value whose type does not match our choice for α. Suppose we choose $\alpha = \mathsf{int}$. This type checks in $\lambda^{?}_{\to}$ because int is consistent with ?, but if the function is called with a boolean argument, a run-time cast error occurs.

The problem with the well-typed-after-substitution approach is that it can "cheat" by assigning ? to a type variable and thereby allow programs to type check that should not. Thus, we need to prevent the type system from adding in arbitrary ?s. On the other hand, we need to allow the propagation of ?s that are already in program annotations.

4. A TYPE SYSTEM FOR $\lambda^{?\alpha}_{\to}$

Loosely speaking, we say that types with more question marks are less informative than types with fewer question marks. The main idea of our new type system is to require the solution for a type variable to be as informative as any type that constrains the type variable. This prevents a solution for a variable from introducing dynamic types that do not already appear in program annotations. Formally, information over types is characterized by the *less or equally informative* relation, written $\sqsubseteq$. This relation is just the partial order underlying the $\sim$ relation[1]. An inductive definition of $\sqsubseteq$ is given below.

Less or Equally Informative

$$\frac{}{? \sqsubseteq \tau} \qquad \frac{}{\gamma \sqsubseteq \gamma} \qquad \frac{\tau_1 \sqsubseteq \tau_3 \quad \tau_2 \sqsubseteq \tau_4}{\tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}$$

The $\sqsubseteq$ relation is a partial order that forms a semi-lattice with ? as the bottom element and $\sqsubseteq$ extends naturally to substitutions. The $\sqsubseteq$ relation is the inverse of the naive subtyping relation ($<:_n$) of Wadler and Findler [55].

We revisit some examples from Section 3 and show how using the $\sqsubseteq$ relation gives us the ability to separate the good programs from the bad. Recall the following example that should be rejected but was not rejected by the well-typed-after-substitution approach.

```
let f (g:α) = g 1
f 1
```

In our approach, the application of g to 1 introduces the constraint $\mathsf{int} \to \beta_0 \sqsubseteq \alpha$ because g is being used as a function from int to β_0. (β_0 is a fresh variable generated for the result of the application.) Likewise, the application of f to 1 introduces the constraint $\mathsf{int} \to \beta_1 \sqsubseteq \alpha \to \beta_0$ which implies $\mathsf{int} \sqsubseteq \alpha$. There is no solution for α that satisfies both $\mathsf{int} \to \beta_0 \sqsubseteq \alpha$ and $\mathsf{int} \sqsubseteq \alpha$, so the program is rejected.

In the next example, the only solution for α should be int.

```
let x:α = x + 1
```

Indeed, in our approach we have the constraint $\mathsf{int} \sqsubseteq \alpha$ whose only solution is $\alpha = \mathsf{int}$.

In the third example, the type system should allow $\alpha = ?$ as a solution.

```
let f (x:?) =
    let y:α = x in y
```

Indeed, we have the constraint $? \sqsubseteq \alpha$, which allows $\alpha = ?$ as a solution. In this case the type system allows many solutions, some of

[1]Each relation is definable in terms of the other: we have $\tau_1 \sim \tau_2$ iff there is a τ_3 such that $\tau_1 \sqsubseteq \tau_3$ and $\tau_2 \sqsubseteq \tau_3$, and in the other direction, $\tau_1 \sqsubseteq \tau_2$ iff for any τ_3, $\tau_2 \sim \tau_3$ implies $\tau_1 \sim \tau_3$.

which, as discussed in Section 3 may introduce unnecessary casts. In our design, the inference algorithm is responsible for choosing a solution that does not introduce unnecessary casts. It will do this by choosing the least informative solution allowed by the type system. This means the inference algorithm chooses the least upper bound of all the types that constraint a type variable as the solution for that variable.

The following program further illustrates how the $\sqsubseteq$ relation constrains the set of valid solutions.

```
let f (g:?→int) (h:int→?) = ...
let k (y:α) = f y y
```

The parameter y is annotated with type variable α and is used in two places, one that expects $? \to \mathsf{int}$ and the other that expects $\mathsf{int} \to ?$. So we have the constraints $? \to \mathsf{int} \sqsubseteq \alpha$ and $\mathsf{int} \to ? \sqsubseteq \alpha$ and the solution is $\alpha = \mathsf{int} \to \mathsf{int}$.

Constraints on type variables can also arise from constraints on compound types that contain type variables. For example, in the following program, we need to delve under the function type to uncover the constraint that $\mathsf{int} \sqsubseteq \alpha$.

```
let g (f:int→int) = f 1
let h (f:α→α) = g f
```

In the next subsection we define how this works in our type system.

4.1 Consistent-equal and Consistent-less

To formalize the notions of constraints between arbitrary types, we introduce two relations: consistent-equal, which has the form $S \models \tau \simeq \tau$ and consistent-less, which has the form $S \models \tau \sqsubseteq \tau$. The two relations are inductively defined in Figure 5. The consistent-equal relation is similar to the type consistency relation $\sim$ except that $\simeq$ gives special treatment to variables. When a variable occurs on either side of the $\simeq$, the type given by S for that variable is required to be at least as informative as the type on the other side according to the consistent-less relation. The consistent-less relation is similar to the $\sqsubseteq$ relation except that it also gives special treatment to variables. When a variable appears on the left, the substitution for that variable is required to be equal to the type on the right. (There is some asymmetry in the $S \models \tau \sqsubseteq \tau$ relation. The substitution is applied to the type on the left and not the right because the substitution has already been applied to the type on the right.)

We illustrate the rules for consistent-equal and consistent-less with the following example.

$$S \models \mathsf{int} \to \alpha \simeq ? \to (\beta \to (\mathsf{int} \to ?))$$

What choices for S satisfies the above constraint? Applying the inverse of the (CEFUN) rule we have

$$S \models \mathsf{int} \simeq ?, \quad S \models \alpha \simeq \beta \to (\mathsf{int} \to ?)$$

The first constraint is satisfied by any substitution using rule (CEDR), but the second constraint is satisfied when

$$S \models \beta \to (\mathsf{int} \to ?) \sqsubseteq S(\alpha)$$

using rule (CEVL). There are many choices for α, but whichever choice is made restricts the choices for β. Suppose

$$S(\alpha) = (? \to \mathsf{bool}) \to (\mathsf{int} \to \mathsf{bool})$$

Then we have

$$S \models \beta \to (\mathsf{int} \to ?) \sqsubseteq (? \to \mathsf{bool}) \to (\mathsf{int} \to \mathsf{bool})$$

and working backwards using rule (CLFUN) yields

$$S \models \beta \sqsubseteq ? \to \mathsf{bool}, \quad S \models \mathsf{int} \to ? \sqsubseteq \mathsf{int} \to \mathsf{bool}$$

$$\boxed{S \models \tau \simeq \tau}$$

(CEG) $$\frac{}{S \models \gamma \simeq \gamma}$$

(CEDL/R) $$\frac{}{S \models ? \simeq \tau} \qquad \frac{}{S \models \tau \simeq ?}$$

(CEFUN) $$\frac{S \models \tau_1 \simeq \tau_3 \quad S \models \tau_2 \simeq \tau_4}{S \models \tau_1 \to \tau_2 \simeq \tau_3 \to \tau_4}$$

(CEVL/R) $$\frac{S \models \tau \sqsubseteq S(\alpha)}{S \models \alpha \simeq \tau} \qquad \frac{S \models \tau \sqsubseteq S(\alpha)}{S \models \tau \simeq \alpha}$$

$$\boxed{S \models \tau \sqsubseteq \tau}$$

(CLVAR) $$\frac{S(\alpha) = \tau}{S \models \alpha \sqsubseteq \tau}$$

(CLG) $$\frac{}{S \models \gamma \sqsubseteq \gamma}$$

(CLDL) $$\frac{}{S \models ? \sqsubseteq \tau}$$

(CLFUN) $$\frac{S \models \tau_1 \sqsubseteq \tau_3 \quad S \models \tau_2 \sqsubseteq \tau_4}{S \models \tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}$$

Figure 5: The consistent-equal and consistent-less relations.

The second constraint is satisfied by any substitution using (CLFUN), (CLG), and (CLDL), but the first constraint is only satisfied when

$$S(\beta) = (? \to \mathsf{bool})$$

according to rule (CLVAR).

A key property of the consistent-equal relation is that it allows the two types to differ in places where they contain ?, but if both sides are variables, then their solutions must be equal, i.e., if $S \models \alpha \simeq \beta$ then $S(\alpha) = S(\beta)$. This is why $\{\alpha \mapsto \mathsf{int}\}$ is a solution for the following program but $\{\alpha \mapsto ?\}$ is not.

```
let f(x:α) =
  let y:β = x in y + 1
```

PROPOSITION 1. *(Properties of $S \models \tau \simeq \tau$ and $S \models \tau \sqsubseteq \tau$)*

1. *$S \models \tau_1 \sqsubseteq \tau_2$ and $S \models \tau_3 \sqsubseteq \tau_2$ implies $S \models \tau_1 \simeq \tau_3$.*
2. *Suppose τ_1 and τ_3 do not contain ?s. Then $S \models \tau_1 \sqsubseteq \tau_2$ and $S \models \tau_1 \simeq \tau_3$ implies $S \models \tau_3 \sqsubseteq \tau_2$.*
3. *If τ_1 and τ_2 contain no ?s and $S \models \tau_1 \simeq \tau_2$, $S(\tau_1) = S(\tau_2)$.*
4. *If τ_1 contains no ?s and $S \models \tau_1 \sqsubseteq \tau_2$, $S(\tau_1) = \tau_2$.*
5. *If $S \models \tau_1 \simeq \tau_2 \to \beta$, then either $\tau_1 = ?$ or there exist τ_{11} and τ_{12} such that $\tau_1 = \tau_{11} \to \tau_{12}$, $\tau_{11} \sim S(\tau_2)$, and $\tau_{12} \sqsubseteq S(\beta)$.*
6. *If $\mathrm{FTV}(\tau_1) = \emptyset$ and $\mathrm{FTV}(\tau_2) = \emptyset$, $S \models \tau_1 \simeq \tau_2$ iff $\tau_1 \sim \tau_2$.*
7. *If $\mathrm{FTV}(\tau_1) = \emptyset$, then $S \models \tau_1 \sqsubseteq \tau_2$ iff $\tau_1 \sqsubseteq \tau_2$.*

4.2 The Definition of the Type System

We formalize our new type system in the setting of the gradually typed lambda calculus with the addition of type variables ($\lambda^{?\alpha}_{\to}$). As in $\lambda^{?}_{\to}$, a function parameter that is not annotated is implicitly annotated with the dynamic type. This favors programs that are mostly dynamic. When a program is mostly static, it would be beneficial to instead interpret variables without annotations as being annotated with unique type variables. This option can easily be offered as a command-line compiler flag. For local variable definitions, provided by the let form, we use the opposite default, infering the type of the variable:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \equiv (\lambda x\colon \beta.\ e_2)\ e_1 \qquad (\beta\ \text{fresh})$$

With the consistent-equal relation in hand we are ready to define the type system for $\lambda^{?\alpha}_{\rightarrow}$ with the judgment $S;\Gamma \vdash_g e : \tau$, shown in Figure 6. The crux of the type system is the application rule (GAPP). We considered a couple of alternatives before arriving at this rule. First we tried to borrow the (SAPP) rule of $\lambda^{\alpha}_{\rightarrow}$ (Figure 4) but replace $S(\tau_1) = S(\tau_2 \rightarrow \tau_3)$ with $S \models \tau_1 \simeq \tau_2 \rightarrow \tau_3$:

$$\frac{S;\Gamma \vdash_g e_1 : \tau_1 \quad S;\Gamma \vdash_g e_2 : \tau_2 \quad S \models \tau_1 \simeq \tau_2 \rightarrow \tau_3}{S;\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

This rule is too lenient: τ_3 may be instantiated with ? which allows too many programs to type check. Consider the following program.

$$\lambda f : \mathsf{int} \rightarrow \mathsf{int}.\ \lambda g : \mathsf{int} \rightarrow \mathsf{bool}.\ f\ (g\ 1)$$

The following is a derivation for this program. The problem is that the application $(g\ 1)$ can be given the type ? because $\{\} \models \mathsf{int} \rightarrow \mathsf{bool} \simeq \mathsf{int} \rightarrow ?$. Let Γ_0 and Γ_1 be the environments defined as follows.

$$\Gamma_0 = \{f \mapsto \mathsf{int} \rightarrow \mathsf{int}\}$$
$$\Gamma_1 = \Gamma_0(g \mapsto (\mathsf{int} \rightarrow \mathsf{bool}))$$

Then we have

$$\cfrac{\cfrac{\{\};\Gamma_1 \vdash_g f : \mathsf{int} \rightarrow \mathsf{int} \qquad \cfrac{\begin{matrix}\{\};\Gamma_1 \vdash_g g : \mathsf{int} \rightarrow \mathsf{bool} \\ \{\};\Gamma_1 \vdash_g 1 : \mathsf{int}\end{matrix}}{\{\};\Gamma_1 \vdash_g g\ 1 : ?}}{\cfrac{\{\};\Gamma_1 \vdash_g (f\ (g\ 1)) : \mathsf{int}}{\{\};\Gamma_0 \vdash_g (\lambda g : \mathsf{int} \rightarrow \mathsf{bool}.\ f\ (g\ 1)) : \mathsf{int}}}}{\{\};\vdash_g (\lambda f : \mathsf{int} \rightarrow \mathsf{int}.\ \lambda g : \mathsf{int} \rightarrow \mathsf{bool}.\ f\ (g\ 1)) : \mathsf{int}}$$

The second alternative we explored borrowed the (APP1) and (APP2) rules from $\lambda^{?}_{\rightarrow}$, replacing $\tau_1 \sim \tau_2$ with $S \models \tau_1 \simeq \tau_2$.

$$\frac{S;\Gamma \vdash_g e_1 : ? \quad S;\Gamma \vdash_g e_2 : \tau}{S;\Gamma \vdash_g e_1\ e_2 : ?}$$

$$\frac{S;\Gamma \vdash_g e_1 : \tau_1 \rightarrow \tau_3 \quad S;\Gamma \vdash_g e_2 : \tau_2 \quad S \models \tau_1 \simeq \tau_2}{S;\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

This alternative also accepts too many programs. Consider the following erroneous program: $((\lambda x : \alpha.\ (x\ 1))\ 1)$. With the substitution $\{\alpha \mapsto ?\}$ this program is well-typed using the first application rule for both applications.

The problem with both of the above approaches is that they allow the type of an application to be ?, thereby adding an extra ? that was not originally in the program. We can overcome this problem by leveraging the definition of the consistent-equal relation, particularly with respect to how it treats type variables: it does not allow the solution for a variable to contain more ?s than the types that constrain it. With this intuition we define the (GAPP) rule as follows.

$$\text{(GAPP)} \quad \frac{\begin{matrix}S;\Gamma \vdash_g e_1 : \tau_1 \quad S;\Gamma \vdash_g e_2 : \tau_2 \\ S \models \tau_1 \simeq \tau_2 \rightarrow \beta \quad (\beta\ \text{fresh})\end{matrix}}{S;\Gamma \vdash_g e_1\ e_2 : \beta}$$

The type of the application is expressed using a type variable instead of a metavariable. This subtle change places a more strict requirement on the variable.

$$\text{(GVAR)} \quad \frac{\Gamma(x) = \tau}{S;\Gamma \vdash_g x : \tau} \qquad \boxed{S;\Gamma \vdash_g e : \tau}$$

$$\text{(GCNST)} \quad S;\Gamma \vdash_g c : \mathit{typeof}(c)$$

$$\text{(GAPP)} \quad \frac{\begin{matrix}S;\Gamma \vdash_g e_1 : \tau_1 \quad S;\Gamma \vdash_g e_2 : \tau_2 \\ S \models \tau_1 \simeq \tau_2 \rightarrow \beta \quad (\beta\ \text{fresh})\end{matrix}}{S;\Gamma \vdash_g e_1\ e_2 : \beta}$$

$$\text{(GABS)} \quad \frac{S;\Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{S;\Gamma \vdash_g \lambda x : \tau_1.\ e : \tau_1 \rightarrow \tau_2}$$

Figure 6: The type system for $\lambda^{?\alpha}_{\rightarrow}$.

Let us revisit the previous examples and show how this rule correctly rejects them. For the first example

$$\lambda f : \mathsf{int} \rightarrow \mathsf{int}.\ \lambda g : \mathsf{int} \rightarrow \mathsf{bool}.\ f\ (g\ 1)$$

we have the constraint set

$$\{\mathsf{int} \rightarrow \mathsf{bool} \simeq \mathsf{int} \rightarrow \beta_1,\ \ \mathsf{int} \rightarrow \mathsf{int} \simeq \beta_1 \rightarrow \beta_2\}$$

which does not have a solution because β_1 must be the upper bound of int and bool but there is no such upper bound. The second example, $((\lambda x : \alpha.\ (x\ 1))\ 1)$, gives rise to the following set of constraints

$$\{\alpha \simeq \mathsf{int} \rightarrow \beta_1,\ \alpha \rightarrow \beta_1 \simeq \mathsf{int} \rightarrow \beta_2\}$$

which does not have a solution because α would have to be the upper bound of $\mathsf{int} \rightarrow \beta_1$ and int.

4.3 Properties of the Type System for $\lambda^{?\alpha}_{\rightarrow}$

When there are no type variable annotations in the program, the type system for $\lambda^{?\alpha}_{\rightarrow}$ is equivalent to that of $\lambda^{?}_{\rightarrow}$.

THEOREM 2. *Suppose* $\mathrm{FTV}(\Gamma) = \emptyset$ *and* $\mathrm{FTV}(e) = \emptyset$.

1. *If* $\Gamma \vdash_g e : \tau$, *then* $\exists S\tau'.\ S;\Gamma \vdash_g e : \tau'$.
2. *If* $S;\Gamma \vdash_g e : \tau$, *then* $\exists \tau'.\ \Gamma \vdash_g e : \tau'$ *and* $\tau' \sqsubseteq S(\tau)$.

PROOF. Each is proved by induction on the typing derivations, with the first statement strengthened prior to applying induction. □

The type system for $\lambda^{?\alpha}_{\rightarrow}$ is stronger (accepts strictly fewer programs) than the alternative type system that says there must be a substitution S that makes the program well-typed in $\lambda^{?}_{\rightarrow}$ (Figure 1).

THEOREM 3.

1. *If* $S;\Gamma \vdash_g e : \tau$ *then there is a* τ' *such that* $S(\Gamma) \vdash_g S(e) : \tau'$ *and* $\tau' \sqsubseteq S(\tau)$.
2. *If* $S(\Gamma) \vdash_g S(e) : \tau$ *then it is not always the case that there is a* τ' *such that* $S;\Gamma \vdash_g e : \tau'$.

PROOF. 1. The proof is by induction on the derivation of $S;\Gamma \vdash_g e : \tau$. The case for (GAPP) uses Proposition 1, items 2 and 5.

2. Here is a counter example: $(\lambda x : \alpha.\ x\ 1)\ 1$.

□

When there are no ?s in the program, a well-typed $\lambda^{?\alpha}_{\rightarrow}$ program is also well-typed in the completely static type system of $\lambda^{\alpha}_{\rightarrow}$. The contrapositive of this statement says that $\lambda^{?\alpha}_{\rightarrow}$ catches all the type errors that are caught by $\lambda^{\alpha}_{\rightarrow}$.

```
unify($\tau_1$,$\tau_2$) =
  case ($\tau_1$,$\tau_2$) of
    ($\gamma$,$\gamma'$) ⇒ if $\gamma = \gamma'$ then return $\emptyset$ else error
    | ($\alpha$,$\tau$) | ($\tau$,$\alpha$) ⇒
      if $\alpha \in \mathrm{FTV}(\tau)$ then error
      else return $\{\alpha \mapsto \tau\}$
    | ($\tau_1 \to \tau_2$,$\tau_3 \to \tau_4$) ⇒
        $S_1$ := unify($\tau_1$,$\tau_3$);
        $S_2$ := unify($S_1(\tau_2)$,$S_1(\tau_4)$);
        return $S_2 \circ S_1$
    | _ ⇒ error
```

Figure 7: Substitution-based unification.

THEOREM 4. *If $e \in \lambda^{\alpha}_{\to}$ and ($\forall\alpha.\ \Gamma(\alpha) = \tau \implies \tau \in \lambda^{\alpha}_{\to}$) then $S;\Gamma \vdash_g e : \tau$ implies $S;\Gamma \vdash e : \tau$ and $\tau \in \lambda^{\alpha}_{\to}$.*

PROOF. The proof is by induction on the derivation of $S;\Gamma \vdash_g e : \tau$. The case for (GAPP) uses Proposition 1 item 3. □

5. AN INFERENCE ALGORITHM FOR $\lambda^{?\alpha}_{\to}$

The inference algorithm we develop for $\lambda^{?\alpha}_{\to}$ follows a similar outline to that of the algorithm for $\lambda^{\alpha}_{\to}$ we presented in Section 2. We generate a set of constraints from the program and then solve the set of constraints. The main difference is that we generate $\simeq$ constraints instead of type equalities, which requires changes to the constraint solver, i.e., the unification algorithm.

The classic substitution-based unification algorithm, reproduced in Figure 7, is not suitable for solving $\simeq$ constraints. Suppose we have the constraint $\{\alpha \to \alpha \simeq ? \to \mathsf{int}\}$. The unification algorithm would first unify α and ? and substitute ? for α on the other side of the $\to$. But ? is not a valid solution for α according to the consistent-equal relation: it is not the case that $\mathsf{int} \sqsubseteq ?$. The problem with the substitution-based unification algorithm is that it treats the first thing that unifies with a variable as the final solution and eagerly applies substitution. To satisfy the $\simeq$ relation, the solution for a variable must be an upper bound of *all* the types that unify with the variable.

The main idea of our new algorithm is that for each type variable α we maintain a type τ that is a lower bound on the solution of α (i.e. $\tau \sqsubseteq \alpha$). (In contrast, inference algorithms for subtyping maintain both lower and upper bounds [40].) When we encounter another constraint $\alpha \simeq \tau'$, we move the lower bound up to be the least upper bound of τ and τ'. This idea can be integrated with some care into a unification algorithm that does not rely on substitution. The algorithm we present is a variant of Huet's almost linear algorithm [23, 26]. We could have adapted Paterson and Wegman's linear algorithm [35] at the expense of a more detailed and less clear presentation.

5.1 Constraint Generation

The constraint generation judgment has the form $\Gamma \vdash_g e : \tau \mid C$, where C is the set of constraints. The constraint generation rules are given in Figure 8 and are straightforward to derive from the type system (Figure 6). The main change is that the side condition on the (GAPP) rule becomes a generated constraint on the (CAPP) rule. The meaning of a set of these constraints is given by the following definition.

DEFINITION 3. *A set of constraints C is **satisfied** by a substitution S, written $S \models C$, iff for any $\tau_1 \simeq \tau_2 \in C$ we have $S \models \tau_1 \simeq \tau_2$.*

$$\boxed{\Gamma \vdash_g e : \tau \mid C}$$

$$(\text{CVAR})\quad \frac{\Gamma(x) = \tau}{\Gamma \vdash_g x : \tau \mid \{\}}$$

$$(\text{CCNST})\quad \Gamma \vdash_g c : \mathit{typeof}(c) \mid \{\}$$

$$(\text{CAPP})\quad \frac{\begin{array}{c}\Gamma \vdash_g e_1 : \tau_1 \mid C_1 \\ \Gamma \vdash_g e_2 : \tau_2 \mid C_2 \qquad (\beta \text{ fresh}) \\ C_3 = \{\tau_1 \simeq \tau_2 \to \beta\} \cup C_1 \cup C_2\end{array}}{\Gamma \vdash_g e_1\ e_2 : \beta \mid C_3}$$

$$(\text{CABS})\quad \frac{\Gamma(x \mapsto \tau) \vdash_g e : \rho \mid C}{\Gamma \vdash_g \lambda x : \tau.\ e : \tau \to \rho \mid C}$$

Figure 8: The definition of constraint generation for $\lambda^{?\alpha}_{\to}$.

We use one of the previous examples to illustrate constraint generation and, in the next subsection, constraint solving.

$$\lambda f : (? \to \mathsf{int}) \to (\mathsf{int} \to ?) \to \mathsf{int}.\ \lambda y : \alpha.\ f\ y\ y$$

We generate the following constraints from this program.

$$\{(? \to \mathsf{int}) \to (\mathsf{int} \to ?) \to \mathsf{int} \simeq \alpha \to \beta_1, \beta_1 \simeq \alpha \to \beta_2\}$$

Because of the close connection between the type system and constraint generation, it is straightforward to show that the two are equivalent.

LEMMA 3. *Given that $\Gamma \vdash e : \tau \mid C$, $S \models C$ is equivalent to $S;\Gamma \vdash_g e : \tau$.*

PROOF. Both directions are proved by induction on the derivation of the constraint generation. □

5.2 Constraint Solver

Huet's algorithm uses a graph representation for types. For example, the type $\alpha \to (\alpha \to \mathsf{int})$ is represented as the node u in the following graph.

$$\begin{tikzcd} u_{\to} \arrow[r] \arrow[d] & w_{\to} \arrow[dl] \arrow[d] \\ v_{\alpha} & x_{\mathsf{int}} \end{tikzcd}$$

Huet used a graph data structure that conveniently combines node labels and out-edges, called the "small term" approach [39, 23]. Each node is labeled with a type, but the type is small in that it consists of either a ground type such as int or a function type ($\to$) whose parameter and return type are nodes instead of types. For example, the above graph is represented by the following stype function from nodes to shallow types.

$$\begin{array}{ll} \mathsf{stype}(u) = v \to w & \mathsf{stype}(v) = \mathsf{var} \\ \mathsf{stype}(w) = v \to x & \mathsf{stype}(x) = \mathsf{int} \end{array}$$

We sometimes write the stype of a node as a subscript, such as $u_{v \to w}$ and x_{int}. Also, when the identity of a node is not important we sometimes just write the stype label in place of the node (e.g., int instead of x_{int}).

Huet's algorithm uses a union-find data structure [49] to maintain equivalence classes among nodes. The operation $\mathsf{find}(u)$ maps node u to its representative node and performs path compression to speed up later calls to find. The operation $\mathsf{union}(u,v,f)$ merges the classes of u and v. If the argument bound to parameter f is true then u becomes the representative of the merged class. Otherwise, the representative is chosen based on which class contains more elements, which reduces the time complexity of the algorithm.

```
solve($C$) =
  $C$ := copy_dyn($C$)
  for each node $u$ do
    $u$.contains_vars := true
  end for
  while not $C$.empty() do
    $x \simeq y$ := $C$.pop()
    $u$ := find($x$); $v$ := find($y$)
    if $u \neq v$ then
      $(u, v, f)$ := order($u$,$v$)
      union($u$, $v$, $f$)
      case stype($u$) $\simeq$ stype($v$) of
         $u_1 \to u_2 \simeq v_1 \to v_2 \Rightarrow$ (* case 1 *)
            $C$.push($u_1, v_1$); $C$.push($u_2, v_2$)
        | $u_1 \to u_2 \simeq ? \Rightarrow$ (* case 2 *)
            if $u$.contains_vars then
              $u$.contains_vars := false
              $w_1$ = vertex(stype=?, contains_vars=false)
              $w_2$ = vertex(stype=?, contains_vars=false)
              $C$.push($w_1 \simeq u_1$); $C$.push($w_2 \simeq u_2$)
        | $\tau \simeq$ var | $\tau \simeq ? \Rightarrow$ (* pass, case 3 and 4 *)
        | $\gamma \simeq \gamma \Rightarrow$ (* pass, case 5 *)
        | _ $\Rightarrow$ error: inconsistent types (* case 6 *)
  end while
  $G$ = the quotient of the graph by equivalence class
  if $G$ is acyclic then
    return $\{u \mapsto$ stype(find($u$)) | $u$ a node in the graph$\}$
  else error

order($u$,$v$) = case stype($u$) $\simeq$ stype($v$) of
                 | $? \simeq \alpha \Rightarrow (u, v, \text{true})$
                 | $? \simeq \tau$ | $\alpha \simeq \tau \Rightarrow (v, u, \text{true})$
                 | $\tau \simeq \alpha \Rightarrow (u, v, \text{true})$
                 | _ $\Rightarrow (u, v, \text{false})$
```

Figure 9: The constraint solving algorithm.

The definition of our solve algorithm is in Figure 9. We defer discussion of the copy_dyn used on the first line. In each iteration of the algorithm we remove a constraint from C, map the pair of nodes x and y to their representatives u and v, and then perform case analysis on the small types of u and v. In each case we merge the equivalence classes for the two nodes and possibly add more constraints. The main difference from Huet's algorithm is some special handling of ?s. When we merge two nodes, we need to decide which one to make the representative and thereby decide which label overrides the other. In Huet's algorithm, a type variable (here nodes labeled var) is overridden by anything else. To handle ?s, we use the rules that ? overrides var but is overridden by anything else. Thus, ? nodes are treated like type variables in that they may merge with any other type. But they are not exactly like type variables in that they override normal type variables. These rules are carried out in cases 3 and 4 of the algorithm.

Before discussing the corner cases of the algorithm, copy_dyn and case 2, we apply the algorithm to the running example introduced in Section 5.1. Figure 10 shows a sequence of snapshots of the solver. Snapshot (a) shows the result of converting the generated constraints to a graph. Constraints are represented as undirected double-lines. At each step, we use bold double-lines to indicate the constraints that are about to be eliminated. To get from (a) to (b) we decompose the constraint between the two function types. Nodes that are no longer the representative of their equivalence class are not shown in the graph. Next we process the two constraints on the left, both of which connect a variable to a function type. The function type becomes the representative in both cases, giving us snapshot (c). As before we decompose a constraint between the two function types into constraints on their children and we have snapshot (d). We first merge the variable node for β_2 into the int node to get (e) and then decompose the constraint between the function type nodes into two more constraints in (f). Here we have constraints on nodes labeled with the ? type. In both cases the node labeled int overrides ? and becomes the representative. The final state is shown in snapshot (g), from which the solutions for the type variables can be read off. As expected, we have $\alpha = \text{int} \to \text{int}$.

Case 2 of the algorithm, for $? \simeq v_1 \to v_2$, deserves some explanation. Consider the program $(\lambda f : ?.\ \lambda x : \alpha.\ f\ x)$. The set of constraints generated from this is $\{? \simeq \alpha \to \beta\}$. According to the operational semantics from Siek and Taha [47], f is cast to $? \to ?$, so in some sense, we really should have the constraint $? \to ? \simeq \alpha \to \beta$. To simulate this in the algorithm we insert two constraints: $? \simeq v_1$ and $? \simeq v_2$. Now, some care must be taken to prevent infinite loops. Consider the constraint $? \simeq v$ where $\text{stype}(v) = v \to v$. The two new constraints are identical to the original. To avoid this problem we mark each node to indicate whether it may contain a variable. The flags are initialized to true and when we see the constraint $? \simeq v$ we change the flag to false.

The copy_dyn function replaces each node labeled ? with a new node labeled ?, thereby removing any sharing of ? nodes. This is necessary to allow certain programs to type check, such as the example in Section 3 with the functions f, g, and h. The following is a simplified example that illustrates the same problem.

$$\lambda f : \text{int} \to \text{bool} \to \text{int}.\ \lambda x :?.\ f\ x\ x$$

From this program we get the constraint set

$$\{\text{int} \to \text{bool} \to \text{int} \simeq\ u_? \to v,\ v \simeq\ u_? \to w\}$$

If we forgo the copy_dyn conversion and just run the solver, we ultimately get int $\simeq\ u_?$ and bool $\simeq\ u_?$ which will result in an error. With the copy_dyn conversion, the two occurrences of $u_?$ are replaced by separate nodes that can separately unify with int and bool and avoid the error. It is important that we apply the copy_dyn conversion to the generated constraints and not to the original program, as that would not avoid the above problem.

The infer function, defined in the following, is the overall inference algorithm, combining constraint generation and solving.

DEFINITION 4. *(Inference algorithm) Given* Γ *and* e*, let* τ*,* C*, and* S *be such that* $\Gamma \vdash e : \tau \mid C$ *and* $S = \text{solve}(C)$*. Then* $\text{infer}(\Gamma, e) = (S, S(\tau))$.

5.3 Properties of the Inference Algorithm

The substitution S returned from the solver is not idempotent. It can be turned into an idempotent substitution by applying it to itself until a fixed point is reached, which we denote by S^*. Note that the solution S' returned by solve is less or equally informative than the other solutions, thereby avoiding types that would introduce unnecessary cast errors.

LEMMA 4. *(Soundness and completeness of the solver)*

1. *If* $S = \text{solve}(C)$ *then* $S^* \models C$.

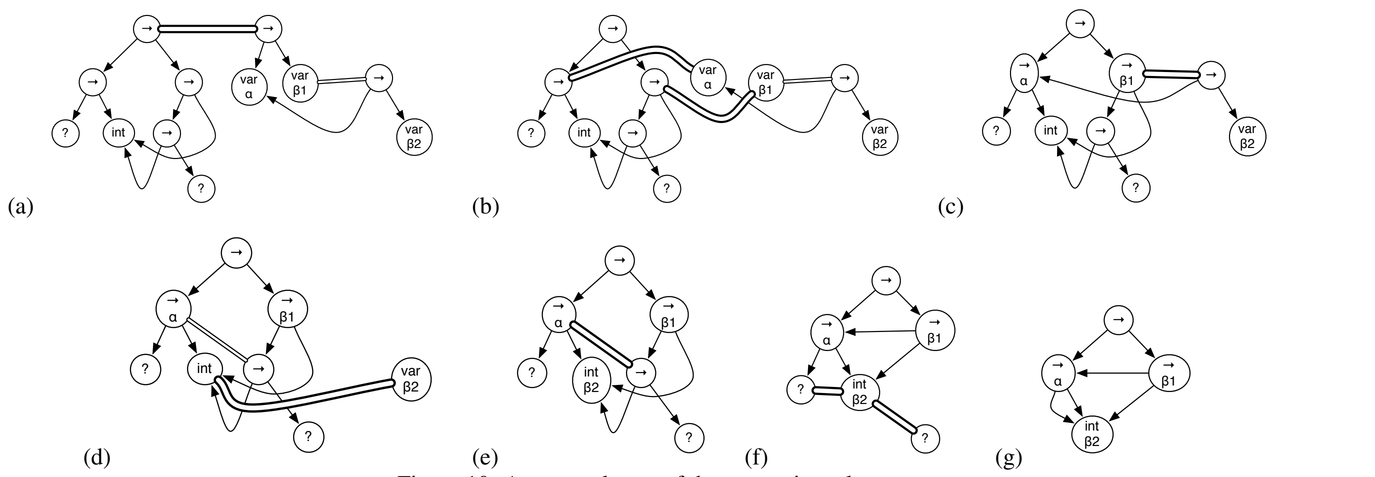

Figure 10: An example run of the constraint solver.

2. *If $S \models C$ then $\exists S' R.\ S' = \mathsf{solve}(C)$ and $R \circ S'^* \sqsubseteq S$.*

PROOF. The correctness of the algorithm is based on the following invariant. Let C be the original set of constraints and C' the set of constraints at a given iteration of the algorithm. At each iteration of the algorithm, $S \models C$ if an only if

1. $S \models C'$,
2. for every pair of type variables α and β in the same equivalence class, $S(\alpha) = S(\beta)$, and
3. there is an R such that $R \circ S' \sqsubseteq S$, where S' is the current solution based on the stype and union-find data structures.

When the algorithm starts, $C = C'$, so the invariant holds trivially. The invariant is proved to hold at each step by case analysis. Once the algorithm terminates, we read off the answer based on the stype and the union-find data structure. This gives a solution that is less informative but more general (in the Hindley-Milner sense) than any other solution, expressed by the clause $R \circ S'^* \sqsubseteq S$. □

LEMMA 5. *The time complexity of the* solve *algorithm is $O(m\alpha(n))$, where n is the number of nodes and m is the number of edges.*

PROOF. The number of iterations in the solve algorithm is $O(m)$. In case 1 of the algorithm we push two constraints into C and make the v node and its two out-edges inaccessible from the find operation. In case 2 of the algorithm, we push two constraints into C and we mark the function type node as no-longer possibly containing variables, which makes it and its two out-edges inaccessible to subsequent applications of case 2. Each iteration performs union-find operations, which have an amortized cost of $\alpha(n)$ [49], so the overall time complexity is $O(m\alpha(n))$. □

THEOREM 5. *(Soundness and completeness of inference)*

1. *If $(S, \tau) = \mathsf{infer}(\Gamma, e)$, then $S^*; \Gamma \vdash_g e : \tau$.*
2. *If $S; \Gamma \vdash_g e : \tau$ then there is a S', τ', and R such that $(S', \tau') = \mathsf{infer}(\Gamma, e)$, $R \circ S'^* \sqsubseteq S$, and $R \circ S'^*(\tau') \sqsubseteq S(\tau)$.*

PROOF. Let τ' and C be such that $\Gamma \vdash e : \tau' | C$.

1. By the soundness of solve (Lemma 4) we have $S^* \models C$. Then by the equivalence of constraint generation and the type system (Lemma 3), we have $S^*; \Gamma \vdash e : \tau$.
2. By the equivalence of constraint generation and the type system (Lemma 3), we have $S \models C$. Then by the completeness of solve (Lemma 4) there exists S' and R such that $S' = \mathsf{solve}(C)$ and $R \circ S'^* \sqsubseteq S$. We then conclude using the definition of infer.

□

THEOREM 6. *The time complexity of the* infer *algorithm is $O(n\alpha(n))$ where n is the size of the program.*

PROOF. The constraint generation step is $O(n)$ and the solver is $O(n\alpha(n))$ (the number of edges in the type graph is bounded by $2n$ because no type has out-degree greater than 2) so the overall time complexity is $O(n\alpha(n))$. □

6. RELATED WORK

The interface between dynamic and static typing has been a fertile area of research.

Optional Types in Dynamic Languages Many dynamic languages allow explicit type annotations such as Common LISP [24], Dylan [45, 11], Cecil [7], Boo [9], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [29], the Bigloo [44, 5] dialect of Scheme [25], and the Strongtalk dialect of Smalltalk [3, 4]. In these languages, adding type annotations brings some static checking and/or improves performance, but the languages do not make the guarantee that annotating all parameters in the program prevents all type errors and type exceptions at run-time, as is the case for gradual typing. One of the primary motivations for the work on gradual typing was to provide a theoretical foundation for optional type annotations in dynamic languages.

Interoperability Gray, Findler, and Flatt [17] consider the problem of interoperability between Java and Scheme and extended Java with a Dynamic type with implicit casts. They did not provide an account of the type system, but their work provided inspiration for our work on gradual typing. Matthews and Findler [28] define an operational semantics for multi-language programs but require programmers to insert explicit "boundary" markers between the two languages, reminiscent of the explicit injection and projections of Abadi et al.

Tobin-Hochstadt and Felleisen [51, 52] developed a system that provides convenient inter-language migration between dynamic and static languages on a per-module basis. In contrast, our goal is to allow migration at finer levels of granularity and to allow for partially typed code. Tobin-Hochstadt and Felleisen build blame tracking into their system and show that run-time type errors may not originate from statically typed modules. As discussed in Section 2.1, gradual typing enjoys a similar property but at a finer level of granularity: run-time type errors do not originate from "safe" regions.

Type Inference There is a huge body of literature on the topic of type inference, especially regarding variations of the Hindley-Milner type system [30, 8, 22]. Of that, the closest to our work is that on combining inference and subtyping [10, 40]. The main difference between inference for subtyping versus gradual typing is that subtyping has co/contra-variance in function types, whereas the consistency relation is covariant in both the parameter and return type, making the inference problem for gradual typing more tractable.

Gradual Typing In addition to the related work discussed in the introduction, we mention a couple more related works here. Anderson and Drossopoulou developed a gradual type system for BabyJ [2] that uses nominal types. Gronski, Knowles, Tomb, Freund, and Flanagan [18] provide gradual typing in the Sage language by including a Dynamic type and implicit down-casts. They use a modified form of subtyping to provide the implicit down-casts.

Quasi-static Typing Thatte's Quasi-Static Typing [50] is close to gradual typing but relies on subtyping and treats the unknown type as the top of the subtype hierarchy. Siek and Taha [47] show that implicit down-casts combined with the transitivity of subtyping creates a fundamental problem that prevents this type system from catching all type errors even when all parameters in the program are annotated.

Riely and Hennessy [42] define a partial type system for Dπ, a distributed π-calculus. Their system allows some locations to be untyped and assigns such locations the type lbad. Their type system, like Quasi-Static Typing, relies on subtyping, however they treat lbad as "bottom", which allows objects of type lbad to be implicitly coercible to any other type.

Soft Typing Static analyses based on dataflow can be used to perform static checking and to optimize performance. The later variant of Soft Typing by Flanagan and Felleisen [14] is an example of this approach. These analyses provide warnings to the programmer while still allowing the programmer to execute their program immediately (even programs with errors), thereby preserving the benefits of dynamic typing. However, the programmer does not control which portions of a program are statically checked: these whole-program analyses have non-local interactions. Gradual typing, in contrast gives the programmer precise control over where static checking is performed.

Dynamic Typing in Statically Typed Languages Abadi et al. [1] extended a statically typed language with a Dynamic type and explicit injection (dynamic) and projection operations (typecase). Their approach does not satisfy the goals of gradual typing, as migrating code between dynamic and static checking not only requires changing type annotations on parameters, but also adding or removing injection and projection operations throughout the code. Gradual typing automates the latter.

Hybrid Typing The Hybrid Type Checking of Flanagan et al. [12, 15] combines standard static typing with refinement types, where the refinements may express arbitrary predicates. This is analogous to gradual typing in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. A notable difference is that hybrid typing is based on subtyping whereas gradual typing is based on type consistency.

Ou et al. [34] define a language that combines standard static typing with more powerful dependent typing. Implicit coercions are allowed to and from dependent types and run-time checks are inserted. This combination of a weaker and a stronger type system is again analogous to gradual typing.

7. CONCLUSION

This paper develops a type system for the gradually typed lambda calculus with type variables ($\lambda^{?\alpha}_{\rightarrow}$). The system integrates unification-based type inference and gradual typing to aid programmers in adding types to their programs. In the proposed system, a programmer uses a type variable annotation to request the best solution for the variable from the inference algorithm.

The type system presented satisfies the defining properties of a gradual type system. That is, a programmer may omit type annotations on function parameters and immediately run the program; run-time type checks are performed to preserve type safety. Furthermore, a programmer may add type annotations to increase static checking. When all function parameters are annotated, all type errors are caught at compile-time.

The paper also develops an efficient inference algorithm for $\lambda^{?\alpha}_{\rightarrow}$ that is sound and complete with respect to the type system and that takes care not to infer types that would introduce cast errors.

As future work, the authors intend to extend the type system and inference algorithm to handle let-polymorphism as it appears in the Hindley-Milner type system. We expect that our unification algorithm can be used as-is in that setting though we expect that the specification of the type system will be rather subtle. We also plan to investigate improving the modularity of the type inference system as it currently performs inference on a whole-program basis.

Acknowledgments

We thank Ronald Garcia, Christoph Reichenbach, Amer Diwan, and Walid Taha for discussions and feedback regarding this paper. This work was funded NSF grant CCF-0702362.

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82, pages 53–81. Elsevier, 2003.

[3] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[4] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

[5] Y. Bres, B. P. Serpette, and M. Serrano. Compiling scheme programs to .NET common intermediate language. In *2nd International Workshop on .NET Technologies*, Pilzen, Czech Republic, May 2004.

[6] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[7] C. Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[9] R. B. de Oliveira. The Boo programming language. http://boo.codehaus.org, 2005.

[10] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[11] N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[12] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[13] C. Flanagan and M. Felleisen. Componential set-based analysis. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1997. ACM Press.

[14] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

[15] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL/WOOD '06: International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[16] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (3rd ed.)*. MIT Press, Cambridge, MA, USA, 2008.

[17] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

[18] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In R. Findler, editor, *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[19] J. J. Heiss. Meet Peter von der Ahé, tech lead for Javac at Sun Microsystems. *Sun Developer Network (SDN)*, April 2007.

[20] D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52, New York, NY, USA, 2007. ACM.

[21] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[22] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans AMS*, 146:29–60, 1969.

[23] G. Huet. *Resolution d'equations dans les langages d'ordre 1, 2, ..., omega.* PhD thesis, Université Paris VII, France, 1976.

[24] G. L. S. Jr. An overview of COMMON LISP. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.

[25] R. Kelsey, W. Clinger, and J. R. (eds.). Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[26] K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, 1989.

[27] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.

[28] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

[29] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[30] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[31] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[32] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3):299–318, 1999.

[33] A. Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994.

[34] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types (extended abstract). In *3rd IFIP International Conference on Theoretical Computer Science*, August 2004.

[35] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[36] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*, December 2002.

[37] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[38] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[39] F. Pottier. A framework for type inference with subtyping. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 228–238, New York, NY, USA, 1998. ACM Press.

[40] F. Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001.

[41] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[42] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–104, New York, NY, USA, 1999. ACM Press.

[43] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[44] M. Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

[45] A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[46] J. Siek and M. Vachharajani. Gradual typing with unification-based inference. Technical Report CU-CS-1039-08, University of Colorado at Boulder, January 2008.

[47] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[48] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LCNS*, pages 2–27. Springer Verlag, August 2007.

[49] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[50] S. Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

[51] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA'06 Companion*, pages 964–974, NY, 2006. ACM.

[52] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL 2008: The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2008.

[53] G. van Rossum. *Python Reference Manual*. Python Software Foundation, `http://docs.python.org/ref/ref.html`, 2.5 edition, September 2006.

[54] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[55] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In D. Dube, editor, *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

[56] B. Wagner. Local type inference, anonymous types, and var. *Microsoft Developer Network*, 2007.

[57] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informatica*, 10:115–122, 1987.