# Gradual Typing for Objects

Jeremy Siek[1] and Walid Taha[2]
jeremy.siek@colorado.edu, taha@rice.edu

[1] University of Colorado, Boulder, CO 80309, USA
and LogicBlox Inc., Atlanta, GA 30309, USA
[2] Rice University, Houston, TX 77005, USA

**Abstract.** Static and dynamic type systems have well-known strengths and weaknesses. In previous work we developed a *gradual type system* for a functional calculus named $\lambda^{?}_{\rightarrow}$. Gradual typing provides the benefits of both static and dynamic checking in a single language by allowing the programmer to control whether a portion of the program is type checked at compile-time or run-time by adding or removing type annotations on variables. Several object-oriented scripting languages are preparing to add static checking. To support that work this paper develops $\mathbf{Ob}^{?}_{<:}$, a gradual type system for object-based languages, extending the $\mathbf{Ob}_{<:}$ calculus of Abadi and Cardelli. Our primary contribution is to show that gradual typing and subtyping are orthogonal and can be combined in a principled fashion. We also develop a small-step semantics, provide a machine-checked proof of type safety, and improve the space efficiency of higher-order casts.

## 1 Introduction

Static and dynamic typing have complementary strengths, making them better for different tasks and stages of development. Static typing provides full-coverage error detection, efficient execution, and machine-checked documentation whereas dynamic typing enables rapid development and fast adaptation to changing requirements. *Gradual typing* allows a programmer to mix static and dynamic checking in a program and provides a convenient way to control which parts of a program are statically checked. The goals for gradual typing are:

- Programmers may omit type annotations on parameters and immediately run the program; run-time type checks are performed to preserve type safety.
- Programmers may add type annotations to increase static checking. When all parameters are annotated, *all* type errors are caught at compile-time.[3]
- The type system and semantics should minimize the implementation burden on language implementors.

In previous work we introduced gradual typing in the context of a functional calculus named $\lambda^{?}_{\rightarrow}$ [47]. This calculus extends the simply typed lambda calculus

[3] The language under study does not include arrays so the claim that we catch all type errors does not include the static detection of out-of-bound errors.

with a statically unknown (dynamic) type ? and replaces type equality with type consistency to allow for implicit coercions that add and remove ?s.

Developers of the object-oriented scripting languages Perl 6 [49] and JavaScript 4 [27] expressed interest in our work on gradual typing. In response, this paper develops the type theoretic foundation for gradual typing in object-oriented languages. Our work is based on the $\mathbf{Ob}^{<:}$ calculus of Abadi and Cardelli, a statically-typed object calculus with structural subtyping. We develop an extended calculus, named $\mathbf{Ob}^{?}_{<:}$, that adds the type ? and replaces the use of subtyping with a relation that integrates subtyping with type consistency.

The boundary between static and dynamic typing is a fertile area of research and the literature addresses many goals that are closely related to those we outline above. Section 8 describes the related work in detail.

The paper starts with a programmer's and an implementor's tour of gradual typing (Sections 2 and 3 respectively) before proceeding with the technical development of the new results in Sections 4, through 7.

*Technical Contributions* This paper includes the following original contributions:

1. The primary contribution of this paper shows that type consistency and subtyping are orthogonal and can be naturally superimposed (Section 4).
2. We develop a syntax-directed type system for $\mathbf{Ob}^{?}_{<:}$ (Section 5).
3. We define a semantics for $\mathbf{Ob}^{?}_{<:}$ via a translation to the intermediate language with explicit casts $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ for which we define a small-step operational semantics (Section 6).
4. We improve the space efficiency of the operational semantics for higher-order casts by applying casts in a lazy fashion to objects (Section 6).
5. We prove that $\mathbf{Ob}^{?}_{<:}$ is type safe (Section 7). The proof is a streamlined variant of Wright and Felleisen's syntactic approach to type soundness [5, 53]. The formalization and proof are based on a proof of type safety for $\mathbf{FOb}^{?}_{<:}$ (a superset of $\mathbf{Ob}^{?}_{<:}$ that also includes functions) we wrote in the Isar proof language [52] and checked using the Isabelle proof assistant [39]. The formalization for $\mathbf{FOb}^{?}_{<:}$ is available in a technical report [46].
6. We prove that $\mathbf{Ob}^{?}_{<:}$ is statically type safe for fully annotated programs (Section 7), that is, we show that neither cast exceptions nor type errors may occur during program execution.

## 2 A Programmer's View of Gradual Typing

We give a description of gradual typing from a programmer's point of view, showing examples in hypothetical variant of the ECMAScript (aka JavaScript) programming language [15] that provides gradual typing. The following Point class definition has no type annotations on the data member x or the dx parameter. The gradual type system therefore delays checks concerning x and dx inside the move method until run-time, as would a dynamically typed language.

```
class Point {
  var x = 0
  function move(dx) { this.x = this.x + dx }
}
var a : int = 1
var p = new Point
p.move(a)
```

More precisely, because the types of the variables x and dx are statically unknown the gradual type system gives them the "dynamic" type, written ? for short. The reader may wonder why we do not infer the type of x from its initializer 0. We discuss the relation between gradual typing and type inference in Section 8. Now suppose the + operator expects arguments of type **int**. The gradual type system allows an *implicit coercion* from type ? to **int**. This kind of coercion could fail (like a down cast) and therefore must be dynamically checked. In statically-typed object-oriented languages, such as Java and C#, implicit up-casts are allowed (they never fail) but not implicit down-casts. Allowing implicit coercions that may fail is *the* distinguishing feature of gradual typing and is what allows gradual typing to support dynamic typing.

To enable the gradual migration of code from dynamic to static checking, gradual typing allows for a mixture of the two and provides seamless interaction between them. In the example above, we define a variable a of type **int**, and invoke the dynamically typed move method. 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 move.

Gradual typing also allows implicit coercions among more complicated types, such as object types. An object type is similar to a Java-style interface in that it contains a list of member signatures, however object types are compared structurally instead of by name. In the following example, the equal method has a parameter o annotated with the object type [x:**int**].

```
class Point {
  var x = 0
  function bool equal(o : [x:int]) { return this.x == o.x }
}
var p = new Point
var q = new Point
p.equal(q)
```

The method invocation p.equal(q) is allowed by the gradual type system. The type of parameter o is [x:**int**] whereas the type of the argument q is [x:?,equal:[x:**int**]→**bool**]. We compare the two types structurally, one member at a time. For x we have a coercion from ? to **int**, so that is allowed. Now consider the equal member. Because this is an object-oriented language with subtyping, we can use an object with more methods in a place that is expecting an object with fewer methods.

Next we look at a fully annotated program, that is, a program where all the variables are annotated with types. In this case the gradual type system acts like a static type system and catches *all* type errors during compilation. In the example below, the invocation of the annotated move method with a string argument is flagged as a static type error.

```
class Point {
  var x : int = 0
  function Point move(dx : int) { this.x = this.x + dx }
}
var p = new Point
p.move("hi") // static type error
```

## 3 An Implementor's View of Gradual Typing

Next we give an overview of gradual typing from a language implementor's point of view, describing the type system and semantics. The main idea of the type system is that we replace the use of type equality with type consistency, written $\sim$. The intuition behind type consistency is to check whether the two types are equal in the parts where both types are known. The following are a few examples. The notation $[l_1 : s_1, \ldots, l_n : s_n]$ is an object type where $l : s$ is the name $l$ and signature $s$ of a method. A signature has the form $\tau \to \tau'$, where $\tau$ is the parameter type and $\tau'$ is the return type of the method.

$$\mathsf{int} \sim \mathsf{int} \qquad \mathsf{int} \not\sim \mathsf{bool} \qquad ? \sim \mathsf{int} \qquad \mathsf{int} \sim ?$$
$$[x : \mathsf{int} \to ?, y : ? \to \mathsf{bool}] \sim [y : \mathsf{bool} \to ?, x : ? \to \mathsf{int}]$$
$$[x : \mathsf{int} \to \mathsf{int}, y : ? \to \mathsf{bool}] \not\sim [x : \mathsf{bool} \to \mathsf{int}, y :? \to \mathsf{bool}]$$
$$[x : \mathsf{int} \to \mathsf{int}, y : ? \to ?] \not\sim [x : \mathsf{int} \to \mathsf{int}]$$

To express the "where both types are known" part of the type consistency relation, we define a restriction operator, written $\sigma|_\tau$. This operator "masks off" the parts of type $\sigma$ that are unknown in type $\tau$. For example,

$$\mathsf{int}|_? = ? \qquad \mathsf{int}|_{\mathsf{bool}} = \mathsf{int}$$
$$[x : \mathsf{int} \to \mathsf{int}, y : \mathsf{int} \to \mathsf{int}]|_{[x:\, ?\to\, ?, y:\mathsf{int}\to\mathsf{int}]} = [x : ? \to ?, y : \mathsf{int} \to \mathsf{int}]$$

The restriction operator is defined as follows.

```
$\sigma|_\tau$ = case $(\sigma, \tau)$ of
        $(-, ?) \Rightarrow ?$
      | $([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_n : t_n]) \Rightarrow$
            $[l_1 : s_1|_{t_1}, \ldots, l_n : s_n|_{t_n}]$
      | $(-, -) \Rightarrow \sigma$

$(\sigma_1 \to \sigma_2)|_{(\tau_1 \to \tau_2)} = (\sigma_1|_{\tau_1}) \to (\sigma_2|_{\tau_2})$
```

**Definition 1.** *Two types $\sigma$ and $\tau$ are* **consistent**, *written $\sigma \sim \tau$, iff $\sigma|_{\tau} = \tau|_{\sigma}$, that is, when the types are equal where they are both known.*[4]

**Proposition 1.** *(Basic Properties of $\sim$)*

1. *$\sim$ is reflexive.*
2. *$\sim$ is symmetric.*
3. *$\sim$ is not transitive. For example,* $\mathsf{bool} \sim ?$ *and* $? \sim \mathsf{int}$ *but* $\mathsf{bool} \not\sim \mathsf{int}$.
4. *$\tau \sim \tau|_{\sigma}$.*
5. *If neither $\sigma$ nor $\tau$ contain ?, then $\sigma \sim \tau$ iff $\sigma = \tau$.*

A gradual type system uses type consistency where a simple type system uses type equality. For example, in the following hypothetical rule for method invocation, the argument and parameter types must be consistent.

$$\frac{\Gamma \vdash e_1 : [\ldots, l : \sigma \to \tau, \ldots] \qquad \Gamma \vdash e_2 : \sigma' \qquad \sigma' \sim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

Gradual typing corresponds to static typing when no ? appear in the program (either explicitly or implicitly) because when neither $\sigma$ nor $\tau$ contain ?, we have $\sigma \sim \tau$ if and only if $\sigma = \tau$, as stated in Proposition 1.

Broadly speaking, there are two ways to implement the run-time behavior of a gradually typed language. One option is to erase the type annotations and interpret the program as if it were dynamically typed. This is an easy way to extend a dynamically typed language with gradual typing. The disadvantages of this approach is that unnecessary run-time type checks are performed and some errors become manifest later in the execution of the program. We do not describe this approach here as it is straightforward to implement.

The second approach performs run-time type checks at the boundaries of dynamically and statically typed code. The advantage is that statically typed code performs no run-time type checks. But there is an extra cost in that run-time tags contain complete types so that objects may be completely checked at boundaries. There are observable differences between the two approaches. The following example runs to completion with the first approach but produces an error with the second approach.

```
function unit foo(dx : int) { }
var x : ? = false; foo(x)
```

In this paper we give a high-level description of the second approach by defining a cast-inserting translation from $\mathbf{Ob}^{?}_{<:}$ to an intermediate language with explicit casts named $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$. The explicit casts have the form $\langle \tau \Leftarrow \sigma \rangle e$, where $\sigma$ is the type of the expression $e$ and $\tau$ is the target type. As an example of cast-insertion, consider the translation of the unannotated move method.

[4] We chose the name "consistency" because it is analogous to the consistency of partial functions. This analogy can be made precise by viewing types as trees and then using the standard encoding of trees as partial functions from tree-paths to labels [41]. The ?s are interpreted as places where the partial function is undefined.

```
  function move(dx) { this.x = this.x + dx }
⇝ function ? move(dx : ?) { this.x = ⟨?⇐ int⟩(⟨int⇐ ?⟩this.x + ⟨int⇐ ?⟩dx) }
```

We define the run-time behavior of $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ with a small-step operational semantics in Section 6. The operational semantics defines rewrite rules that simplify an expression until it is either a value or until it gets stuck (no rewrite rules apply). A stuck expression corresponds to an error. We distinguish between two kinds of errors: *cast errors* and *type errors*. A cast error occurs when the run-time type of a value is not consistent with the target type of the cast. Cast errors can be thought of as triggering exceptions, though for simplicity we do not model exceptions here. We categorize all other stuck expressions as type errors.

**Definition 2.** *A program is* **statically type safe** *when neither cast nor type errors can occur during execution. A program is* **type safe** *when no type errors can occur during execution.*

In Section 7 we show that any $\mathbf{Ob}^{?}_{<:}$ program is *type safe* and that any $\mathbf{Ob}^{?}_{<:}$ program that is fully annotated is *statically type safe.*

## 4 Combining Gradual Typing and Subtyping

In previous work we discovered that approaches to gradual typing based on subtyping and ? as "top" do not achieve *static type safety* for fully annotated terms [47]. The problem is that if you allow an implicit down-cast from "top" to any type (? <: $S$), then you can use the normal up-cast rule $R$ <: ? and transitivity to deduce $R <: S$ for *any* two types $R$ and $S$. The resulting type system therefore accepts all programs and does not reject programs that have static type errors. This discovery led us to the type consistency relation which formed the basis for our gradual type system for functional languages. However, subtyping is a central feature of object-oriented languages, so the question is how can we add subtyping to gradual type system while maintaining static type safety for fully annotated terms? It turns out to be as simple as adding subsumption:

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

We do not treat ? as the top of the subtype hierarchy, but instead treat ? as neutral to subtyping, with only ? <: ?. The following defines subtyping.[5]

$$\mathsf{int} <: \mathsf{int} \qquad \mathsf{float} <: \mathsf{float} \qquad \mathsf{bool} <: \mathsf{bool} \qquad ? <: ?$$

$$\mathsf{int} <: \mathsf{float} \qquad [l_i : s_i \ ^{i\in 1\ldots n+m}] <: [l_i : s_i \ ^{i \in 1\ldots n}]$$

[5] The calculus $\mathbf{Ob}^{?}_{<:}$ does not include functions, so no subtyping rules for function types are provided here. The calculus $\mathbf{FOb}^{?}_{<:}$ in the technical report [46] includes function types.

While the type system is straightforward to define, more care is needed to define 1) a type checking *algorithm* and 2) an operational semantics that takes subtyping into account. In this section we discuss the difficulties in defining a type checking algorithm and present a solution.

It is well known that a type checking algorithm cannot use the subsumption rule because it is inherently non-deterministic. (The algorithm would need to guess when to apply the rule and what target type to use.) Instead of using subsumption, the standard approach is to use the subtype relation in the other typing rules where necessary [41]. The following is the result of applying this transformation to our gradually typed method invocation rule.

$$\frac{\Gamma \vdash e_1 : [\ldots, l : \sigma \to \tau, \ldots] \qquad \Gamma \vdash e_2 : \sigma' \qquad \sigma' <: \sigma'' \qquad \sigma'' \sim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

This rule still contains some non-determinacy because of the type $\sigma''$. We need a combined relation that directly compares $\sigma'$ and $\sigma$.

Fortunately there is a natural way to define a relation that takes both type consistency and subtyping into account. To review, two types are consistent when they are equal where both are known, i.e., $\sigma \sim \tau$ iff $\sigma|_\tau = \tau|_\sigma$. To combine type consistency with subtyping, we replace type equality with subtyping.

**Definition 3 (Consistent-Subtyping).** $\sigma \lesssim \tau \equiv \sigma|_\tau <: \tau|_\sigma$

Here we apply the restriction operator to types $\sigma$ and $\tau$ that may differ according to the subtype relationship, so we must update the definition of restriction to allow for objects of differing widths, as shown below.

$$
\begin{array}{l}
\sigma|_\tau = \textbf{case } (\sigma, \tau) \text{ of} \\
\qquad\qquad (-, ?) \Rightarrow ? \\
\qquad\quad \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m]) \text{ where } n \leq m \Rightarrow \\
\qquad\qquad\quad [l_1 : s_1|_{t_1}, \ldots, l_n : s_n|_{t_n}] \\
\qquad\quad \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m]) \text{ where } n > m \Rightarrow \\
\qquad\qquad\quad [l_1 : s_1|_{t_1}, \ldots, l_m : s_m|_{t_m}, l_{m+1} : s_{m+1}, \ldots, l_n : s_n] \\
\qquad\quad \mid (-, -) \Rightarrow \sigma \\
\\
(\sigma_1 \to \sigma_2)|_{(\tau_1 \to \tau_2)} = (\sigma_1|_{\tau_1}) \to (\sigma_2|_{\tau_2})
\end{array}
$$

The following proposition allows us to replace the conjunction $\sigma' <: \sigma''$ and $\sigma'' \sim \sigma$ with $\sigma' \lesssim \sigma$ in the gradual method invocation rule.

**Proposition 2 (Properties of Consistent-Subtyping).** *The following are equivalent:*

1. $\sigma \lesssim \tau$,
2. $\sigma <: \sigma'$ *and* $\sigma' \sim \tau$ *for some* $\sigma'$, *and*
3. $\sigma \sim \sigma''$ *and* $\sigma'' <: \tau$ *for some* $\sigma''$.

The method invocation rule can now be formulated in a syntax-directed fashion using the consistent-subtyping relation.

$$\frac{\Gamma \vdash e_1 : [\ldots, l : \sigma \to \tau, \ldots] \qquad \Gamma \vdash e_2 : \sigma' \qquad \sigma' \lesssim \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

It is helpful to think of the type consistency and subtyping relation as allowing types to differ along two different axes, with $\sim$ along the x-axis and $<:$ along the y-axis. With this intuition, the following informal diagram represents Proposition 2.

$$\begin{tikzcd}
\sigma' \arrow[rr, dash, bend left, "\sim"] & \tau|_{\sigma} \arrow[r, dash, "\sim"'] & \tau \\
\sigma \arrow[u, "<:"] \arrow[r, dash, "\sim"] \arrow[rr, dash, bend right, "\sim"'] & \sigma|_{\tau} \arrow[u, "<:"] & \sigma'' \arrow[u, "<:"]
\end{tikzcd}$$

The following is an example of the above diagram for a particular choice of types.

$$\begin{tikzcd}
{[x : \mathsf{int} \to ?]} \arrow[rr, dash, bend left, "\sim"] & {[x : ? \to ?]} \arrow[r, dash, "\sim"'] & {[x : ? \to \mathsf{int}]} \\
{[x : \mathsf{int} \to ?, y : \mathsf{bool} \to \mathsf{bool}]} \arrow[u, "<:"] \arrow[r, dash, "\sim"] \arrow[rr, dash, bend right, "\sim"'] & {[x : ? \to ?, y : \mathsf{bool} \to \mathsf{bool}]} \arrow[u, "<:"] & {[x : ? \to \mathsf{int}, y : \mathsf{bool} \to \mathsf{bool}]} \arrow[u, "<:"]
\end{tikzcd}$$

## 5 A Gradually Typed Object Calculus

We define a gradually typed object calculus named $\mathbf{Ob}^{?}_{<:}$ by extending Abadi and Cardelli's $\mathbf{Ob}_{<:}$ [1] with the unknown type ?. For purposes of exposition, we add one parameter (in addition to self) to methods. The syntax of $\mathbf{Ob}^{?}_{<:}$ includes three constructs for working with objects. The form $[l_i{=}\tau_i\ \varsigma(x_i : \sigma_i)e_i\ ^{i\in 1\ldots n}]$ creates an object containing a set of methods. Each method has a name $l_i$, a parameter $x_i$ with type annotation $\sigma_i$, a body $e_i$, and a return type $\tau_i$. The $\varsigma$ symbol just means "method" and is reminiscent of the $\lambda$ used in functional calculi. The self parameter is implicit. Omitting a type annotation is short-hand for annotating with type ?. Multi-parameter methods can be encoded using single-parameter methods [1]. The form $e_1.l(e_2)$ is a method invocation, where $e_1$ is the receiver object, $l$ is the method to invoke, and $e_2$ is the argument. The form $e_1.l{:=}\tau\ \varsigma(x{:}\sigma)e_2$ is a method update. The result is a copy of $e_1$ except that its method $l$ is replaced by the right-hand side. Abadi and Cardelli chose not to represent fields in the core calculus but instead encode fields as methods. The following is an example of a point object in $\mathbf{Ob}^{?}_{<:}$:
[equal=**bool** ς(p:[x:**int**]) self.x.eq(p.x), x=zero].

$$\begin{array}{llcll}
\text{Variables} & x \in \mathbb{X} & \supseteq & \{\mathsf{self}\} & \boxed{e \in \mathbf{Ob}_{<:}} \\
\text{Method labels} & l \in \mathbb{L} & & & \\
\text{Ground Types} & \gamma \in \mathbb{G} & \supseteq & \{\mathsf{bool}, \mathsf{int}, \mathsf{float}, \mathsf{unit}\} & \\
\text{Constants} & c \in \mathbb{C} & \supseteq & \{\mathsf{true}, \mathsf{false}, \mathsf{zero}, 0.0, ()\} & \\
\text{Types} & \rho, \sigma, \tau & ::= & \gamma \mid [l_i : s_i \ ^{i \in 1 \ldots n}] & \\
\text{Method Sig.} & s, t & ::= & \tau \to \tau & \\
\text{Expressions} & e & ::= & x \mid c \mid [l_i{=}\tau_i\ \varsigma(x_i{:}\sigma_i)\ e_i \ ^{i \in 1 \ldots n}] \mid & \\
 & & & e.l(e) \mid e.l{:=}\tau\ \varsigma(x{:}\sigma)e & \\
\text{Syntactic Sugar} & l{=}e : \tau & \equiv & l{=}\tau\ \varsigma(x{:}\mathsf{unit})e \qquad (x \notin e) & \\
 & e.l & \equiv & e.l(()) & \\
 & e_1.l{:=}e_2 : \tau & \equiv & e_1.l{:=}\tau\ \varsigma(x{:}\mathsf{unit})e_2 \qquad (x \notin e) & \\
 & & & & \\
\text{Types} & \rho, \sigma, \tau & {+}{=} & ? & \boxed{e \in \mathbf{Ob}^{?}_{<:} \supset \mathbf{Ob}_{<:}} \\
\text{Syntactic Sugar} & \varsigma(x)e & \equiv & ?\ \varsigma(x{:}?)e & \\
 & l{=}e & \equiv & l{=}e\ {:}? & \\
 & e_1.l{:=}e_2 & \equiv & e_1.l{:=}e_2\ {:}? &
\end{array}$$

The gradual type system for $\mathbf{Ob}^{?}_{<:}$ is shown in Figure 1. (For reference, the type system for $\mathbf{Ob}_{<:}$ is in the Appendix, Fig. 4.) We use the symbol $\Gamma$ for environments, which are finite partial functions from variables to types. The type system is parameterized on a *TypeOf* function that maps constants to types.

There are two rules for each elimination form. The first rule handles the case when the type of the receiver is unknown and the second rule handles when the type of the receiver is known. In the (GIvk1) rule for method invocation, the type of the receiver $e_1$ is unknown and the type of the argument $e_2$ is unconstrained. The rule (GIvk2) is described in Section 4, and is where we use the consistent-subtyping relation $\lesssim$. The rule (GUpd1) for method update handles the case when the type of the receiver $e_1$ is unknown. The new method body is type checked in an environment where self is bound to ? and the parameter $x$ is bound to its declared type $\sigma$. The result type for this expression is $[l : \sigma \to \tau]$. [6] The rule (GUpd2) handles the case for method update when the type of the receiver is an object type $\rho$. The new method body is type checked in an environment where self is bound to $\rho$ and $x$ is bound to its declared type $\sigma$. The constraints $\sigma_k \lesssim \sigma$ and $\tau \lesssim \tau_k$ make sure that the new method can be coerced to the type of the old method.

[6] The result type for (GUpd1) is somewhat unsatisfactory because a method $l' \neq l$ can be invoked on $e$ but not on the updated version of $e$. This can be easily resolved by extending the type system to include open object types in addition to closed object types, as is done in OCaml. If an object has an open object type you may invoke methods that are not listed in its type.

**Fig. 1.** A Gradual Type System for Objects.

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

(GVar)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_G x : \tau}$$

(GConst)
$$\Gamma \vdash_G c : \mathit{TypeOf}(c)$$

(GObj)
$$\frac{\Gamma, \mathsf{self}:\rho, x_i : \sigma_i \vdash_G e_i : \tau_i \qquad \forall i \in 1 \ldots n}{\Gamma \vdash_G [l_i{=}_{\tau_i}\, \varsigma(x_i{:}\sigma_i)e_i \ ^{i\in 1\ldots n}] \ : \rho}$$
$$\text{(where } \rho \equiv [l_i : \sigma_i \to \tau_i \ ^{i \in 1 \ldots n}])$$

(GIvk1)
$$\frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau}{\Gamma \vdash_G e_1.l(e_2) : ?}$$

(GIvk2)
$$\frac{\Gamma \vdash_G e_1 : [\ldots, l : \sigma \to \tau, \ldots] \qquad \Gamma \vdash_G e_2 : \sigma' \qquad \sigma' \lesssim \sigma}{\Gamma \vdash_G e_1.l(e_2) : \tau}$$

(GUpd1)
$$\frac{\Gamma \vdash_G e : ? \qquad \Gamma, \mathsf{self}: ?, x{:}\sigma \vdash e' : \tau}{\Gamma \vdash_G e.l{:=}_{\tau}\, \varsigma(x{:}\sigma)e' : \ [l : \sigma \to \tau]}$$

(GUpd2)
$$\frac{\Gamma \vdash_G e_1 : \rho \qquad \Gamma, \mathsf{self}:\rho, x{:}\sigma \vdash_G e_2 : \tau \qquad \sigma_k \lesssim \sigma \qquad \tau \lesssim \tau_k}{\Gamma \vdash_G e_1.l_k{:=}_{\tau}\, \varsigma(x:\sigma)e_2 : \rho}$$
$$\text{(where } \rho \equiv [l_i : \sigma_i \to \tau_i \ ^{i\in 1\ldots n}] \text{ and } k \in 1 \ldots n)$$

## 6 A semantics for $\mathbf{Ob}^{?}_{<:}$

In this section we define a semantics for $\mathbf{Ob}^{?}_{<:}$ by defining a cast-inserting translation to the intermediate language $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ and by defining an operational semantics for $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$. The syntax and typing rules for the intermediate language are those of $\mathbf{Ob}_{<:}$ [1] (Fig. 4 of the Appendix) extended with an explicit cast. The syntax and typing rule for the explicit cast are shown below.

**Intermediate Language**

Expressions $e \quad \mathrel{+}= \langle \tau \Leftarrow \tau \rangle e$ $\qquad \boxed{e \in \mathbf{Ob}^{\langle\cdot\rangle}_{<:} \supset \mathbf{Ob}_{<:}}$

$$\ldots \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \sim \tau \qquad \sigma \neq \tau}{\Gamma \vdash \langle \tau \Leftarrow \sigma \rangle e : \tau} \qquad \boxed{\Gamma \vdash e : \tau}$$

Most run-time systems for dynamic languages associate a "type tag" with each value so that run-time type checks can be performed efficiently. In this paper we use a term-rewriting semantics that works directly on the syntax, without auxiliary structures. Instead of type tags, the cast expressions themselves are used to support run-time type checking. The cast includes both the source and target type because both pieces of information are needed at run-time to apply casts to objects.

We do not allow “no-op” casts in the intermediate language to simplify the canonical forms of values, e.g., a value of type $\mathsf{int}$ is an integer, and not an integer cast to $\mathsf{int}$. The typing rule for casts requires the source and target type to be consistent, so the explicit cast may only add or remove ?'s from the type. Implicit up-casts due to subtyping remain implicit using a subsumption rule, as such casts are safe and there is no need for run-time checking.

### 6.1 The Cast Insertion Translation

The cast insertion translation is guided by the gradual type system, inserting casts wherever the type of a subexpression differs from the expected type. For example, recall the rule for method invocation.

$$(\textsc{GIvk2})\ \frac{\mathit{\Gamma} \vdash_G e_1 : [\ldots, l : \tau \to \tau', \ldots] \qquad \mathit{\Gamma} \vdash_G e_2 : \sigma \qquad \sigma \lesssim \tau}{\mathit{\Gamma} \vdash_G e_1.l(e_2) : \tau'}$$

The type $\sigma$ of $e_2$ may differ from the method's parameter type $\tau$. We need to translate the invocation to a well typed term of $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$, where the argument type must be a subtype of the parameter type. We know that $\sigma \lesssim \tau$, so $\sigma$ can differ from $\tau$ along both the type consistency relation $\sim$ and the subtype relation $<:$. So we have the diagram on the left:

$$\begin{tikzcd}
 & \tau \\
\sigma \arrow[ur, "\lesssim"] &
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
 & \tau \\
\sigma \arrow[ur, "\lesssim"] \arrow[r, "\sim"'] & \rho \arrow[u, "<:"']
\end{tikzcd}$$

A cast can move us along the x-axis, and the subsumption rule can move us along the y-axis. So a solution to the problem, shown above on the right, is to cast $e_2$ from $\sigma$ to some type $\rho$ where $\rho <: \tau$. (We could just as well move up along the y-axis via subsumption before casting along the x-axis; it makes no difference.) The following example shows how we can choose $\rho$ for a particular situation and gives some intuition for how we can choose it in general.

$$\begin{tikzcd}
 & {[x : ? \to \mathsf{int}]} \\
{[x : \mathsf{int} \to ?, y : \mathsf{bool} \to \mathsf{bool}]} \arrow[ur, "\lesssim"] \arrow[r, "\sim"'] & {[x : ? \to \mathsf{int}, y : \mathsf{bool} \to \mathsf{bool}]} \arrow[u, "<:"']
\end{tikzcd}$$

The type $\rho$ must be the same width (have the same methods) as $\sigma$, and it must have a ? in all the locations that correspond to ?s in $\tau$ (and not have ?s where $\tau$ does not). In general, we can construct $\rho$ with the merge operator, written $\sigma \leftharpoonup \tau$, defined below.

$$\begin{array}{l}
\sigma \leftharpoonup \tau \equiv \textsf{\textbf{case}}\ (\sigma, \tau)\ \textsf{of} \\
\qquad\qquad (?, -) \Rightarrow \tau \\
\qquad\quad\ \mid (-, ?) \Rightarrow\ ? \\
\qquad\quad\ \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m])\ \textsf{where}\ n \leq m \Rightarrow \\
\qquad\qquad\quad [l_1 : s_1 \leftharpoonup t_1, \ldots, l_n : s_n \leftharpoonup t_n] \\
\qquad\quad\ \mid ([l_1 : s_1, \ldots, l_n : s_n], [l_1 : t_1, \ldots, l_m : t_m])\ \textsf{where}\ n > m \Rightarrow \\
\qquad\qquad\quad [l_1 : s_1 \leftharpoonup t_1, \ldots, l_m : s_m \leftharpoonup t_m, l_{m+1} : s_{m+1}, \ldots, l_n : s_n] \\
\qquad\quad\ \mid (-, -) \Rightarrow \sigma \\
\\
(\sigma_1 \to \sigma_2) \leftharpoonup (\tau_1 \to \tau_2) = (\sigma_1 \leftharpoonup \tau_1) \to (\sigma_2 \leftharpoonup \tau_2)
\end{array}$$

With the merge operator, we have the following diagram:

$$\begin{tikzcd}
 & \sigma \\
\sigma' \arrow[ur, "\lesssim"] \arrow[r, "\lesssim"'] & (\sigma' \leftharpoonup \sigma) \arrow[u, "<:"']
\end{tikzcd}$$

**Proposition 3 (Basic Properties of $\leftharpoonup$).**

1. *$(\sigma \leftharpoonup \sigma) = \sigma$*
2. *$\sigma \sim (\sigma \leftharpoonup \tau)$*
3. *If $\sigma \lesssim \tau$ then $(\sigma \leftharpoonup \tau) <: \tau$.*

The cast insertion judgment $\mathit{\Gamma} \vdash e \rightsquigarrow e' : \tau$ translates an expression $e$ in the environment $\mathit{\Gamma}$ to $e'$ and determines that its type is $\tau$. The cast insertion rule for method invocation (on known object types) is defined as follows using $\sigma' \leftharpoonup \sigma$ as the target of the cast on $e_2$.

$$(\text{CI}\textsc{vk}2)\,\frac{\mathit{\Gamma} \vdash e_1 \rightsquigarrow e_1' : [\ldots, l : \sigma \to \tau, \ldots] \qquad \mathit{\Gamma} \vdash e_2 \rightsquigarrow e_2' : \sigma' \qquad \sigma' \lesssim \sigma}{\mathit{\Gamma} \vdash e_1.l(e_2) \rightsquigarrow e_1'.l(\langle\!\langle(\sigma' \leftharpoonup \sigma) \Leftarrow \sigma'\rangle\!\rangle e_2') : \tau}$$

In the case when $\sigma' = \sigma$, we do not insert a cast, which is why we use the following helper function.

$$\langle\!\langle\tau \Leftarrow \sigma\rangle\!\rangle e \equiv \textsf{if}\ \sigma = \tau\ \textsf{then}\ e\ \textsf{else}\ \langle\tau \Leftarrow \sigma\rangle e$$

The rest of the translation rules are straightforward. Fig. 2 gives the full definition of the cast insertion translation.

The cast-insertion judgment subsumes the gradual type system and additionally specifies how to produce the translation. In particular, a cast-insertion derivation can be created for precisely those terms accepted by the type system.

**Proposition 4 (Cast Insertion and Gradual Typing).**
*$\mathit{\Gamma} \vdash_G e : \tau$ iff $\exists e'.\ \mathit{\Gamma} \vdash e \rightsquigarrow e' : \tau$.*

When there is a cast insertion translation for term $e$, the resulting term $e'$ is guaranteed to be a well-typed term of the intermediate language. Lemma 1 is used directly in the type safety theorem.

**Fig. 2.** Cast Insertion

$$\boxed{\Gamma \vdash e \leadsto e' : \tau}$$

(CVar) $$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \leadsto x : \tau}$$

(GConst) $$\Gamma \vdash c \leadsto c : \mathit{TypeOf}(c)$$

(CObj) $$\frac{\Gamma, \mathsf{self} : \rho, x_i : \sigma_i \vdash e_i \leadsto e'_i : \tau_i \qquad \forall i \in 1 \ldots n}{\begin{array}{c}\Gamma \vdash [l_i{=}\tau_i\ \varsigma(x_i : \sigma_i)e_i\ ^{i \in 1 \ldots n}] \leadsto [l_i{=}\tau_i\ \varsigma(x_i : \sigma_i)e'_i\ ^{i \in 1 \ldots n}] : \rho \\ (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i\ ^{i \in 1 \ldots n}])\end{array}}$$

(CIvk1) $$\frac{\Gamma \vdash e_1 \leadsto e'_1 : ? \qquad \Gamma \vdash e_2 \leadsto e'_2 : \tau}{\Gamma \vdash_G e_1.l(e_2) \leadsto (\langle\!\langle [l : \tau \rightarrow ?] \Leftarrow ? \rangle\!\rangle e'_1).l(e'_2) : ?}$$

(CIvk2) $$\frac{\Gamma \vdash e_1 \leadsto e'_1 : [\ldots, l : \sigma \rightarrow \tau, \ldots] \qquad \Gamma \vdash e_2 \leadsto e'_2 : \sigma' \qquad \sigma' \lesssim \sigma}{\Gamma \vdash e_1.l(e_2) \leadsto e'_1.l(\langle\!\langle (\sigma' \leftharpoondown \sigma) \Leftarrow \sigma' \rangle\!\rangle e'_2) : \tau}$$

(CUpd1) $$\frac{\Gamma \vdash e_1 \leadsto e'_1 : ? \qquad \Gamma, \mathsf{self} : ?, x : \sigma \vdash e_2 \leadsto e'_2 : \tau}{\Gamma \vdash e_1.l{:=}\tau\ \varsigma(x : \sigma)e_2 \leadsto (\langle\!\langle [l : \sigma \rightarrow \tau] \Leftarrow ? \rangle\!\rangle e'_1).l{:=}\tau\ \varsigma(x : \sigma)e'_2 :\ [l : \sigma \rightarrow \tau]}$$

(CUpd2) $$\frac{\begin{array}{c}\Gamma \vdash e_1 \leadsto e'_1 : \rho \qquad\qquad \Gamma, \mathsf{self} : \rho, x : \sigma \vdash e_2 \leadsto e'_2 : \tau \\ \sigma_k \lesssim \sigma \qquad \tau \lesssim \tau_k \qquad e_3 \equiv \langle\!\langle \tau_k \Leftarrow \tau \rangle\!\rangle [x \mapsto \langle\!\langle \sigma \Leftarrow \sigma_k \rangle\!\rangle y] e'_2\end{array}}{\begin{array}{c}\Gamma \vdash e_1.l_k{:=}\tau\ \varsigma(x : \sigma)e_2 \leadsto e'_1.l_k{:=}\tau_k\ \varsigma(y : \sigma_k)e_3 : \rho \\ (\text{where } \rho \equiv [l_i : \sigma_i \rightarrow \tau_i\ ^{i \in 1 \ldots n}] \text{ and } k \in 1 \ldots n)\end{array}}$$

**Lemma 1 (Cast Insertion is Sound).**
*If $\Gamma \vdash e \leadsto e' : \tau$ then $\Gamma \vdash e' : \tau$.*

*Proof.* The proof is by induction on the cast insertion derivation. □

The next lemma is needed to prove *static type safety*, that is, a fully annotated term is guaranteed to produce neither cast nor type errors. The set of fully annotated terms of $\mathbf{Ob}^?_{<:}$ is exactly the $\mathbf{Ob}_{<:}$ subset of $\mathbf{Ob}^?_{<:}$. The function FV returns the set of variables that occur free in an expression.

**Lemma 2 (Cast Insertion is the Identity for $\mathbf{Ob}_{<:}$).**
*If $\Gamma \vdash e \leadsto e' : \tau$ and $e \in \mathbf{Ob}_{<:}$ and $\forall x \in \mathrm{FV}(e) \cap \mathrm{dom}(\Gamma).\ \Gamma(x) \in \mathbf{Ob}_{<:}$ then $\Gamma \vdash e : \tau$ and $\tau \in \mathbf{Ob}_{<:}$ and $e = e'$.*

*Proof.* The proof is by induction on the cast insertion derivation. □

Lemma 2 is also interesting for performance reasons. It shows that for fully annotated terms, no casts are inserted so there is no run-time type checking overhead.

### 6.2 Operational Semantics of $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$

In this section we define a small-step, evaluation context semantics [17, 18, 53] for $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$. Evaluation reduces expressions to values.

**Definition 4 (Values and Contexts).** Simple values *are constants, variables, and objects.* Values *are simple values or a simple value enclosed in a single cast. An* evaluation context *is an expression with a hole in it (written* $[]$*) to mark where rewriting (reduction) may take place.*

$$\begin{array}{lll} \textit{Simple Values} & \xi & ::= c \mid x \mid [l_i{=}\tau_i\ \varsigma(x_i\!:\!\sigma_i)e_i\ ^{i\in 1\ldots n}] \\ \textit{Values} & v & ::= \xi \mid \langle\tau \Leftarrow \tau\rangle\xi \\ \textit{Contexts} & E & ::= [] \mid E.l(e) \mid v.l(E) \mid E{:=}\tau\ \varsigma(x\!:\!\tau)e \mid \langle\tau \Leftarrow \tau\rangle E \end{array}$$

The reduction rules are specified in Fig. 3. When a reduction rule applies to an expression, the expression is called a redex:

**Definition 5 (Redex).** $\textit{redex}\, e \equiv \exists e'.\ e \longrightarrow e'$

The semantics is parameterized on a $\delta$-function that defines the behavior of the primitive methods attached to the constants. The rule for method invocation (Ivk) looks up the body of the appropriate method and substitutes the argument for the parameter. The primitive method invocation rule (Delta) simply evaluates to the result of applying $\delta$. In both the (Ivk) and (Delta) rules, the argument is required to be a value as indicated by the use of meta-variable $v$. Method update (Upd) creates a new object in which the specified method has been replaced.

The traditional approach to evaluating casts is to apply them in an eager fashion. For example, casting at function types creates a wrapper function with the appropriate casts on the input and output [19, 20, 21, 48].

$$\langle(\rho \to \nu) \Leftarrow (\sigma \to \tau)\rangle v \longrightarrow (\lambda x\!:\!\rho.\ \langle\nu \Leftarrow \tau\rangle(v\ (\langle\sigma \Leftarrow \rho\rangle x))$$

The problem with this approach is that the wrapper functions can build up, one on top of another, using memory in proportion to the number of cast applications. The solution we use here is to delay the application of casts, and to collapse sequences of casts into a single cast. When a cast is applied to a value that is already wrapped in a cast, either the (Merge) or (Remove) rule applies, or else the cast is a "bad cast".

**Definition 6 (Bad Cast).**

$$\begin{aligned} \textit{badcast}\, e &\equiv \exists v\ \rho\ \sigma\ \sigma'\ \tau.\ e = \langle\tau \Leftarrow \sigma'\rangle\langle\sigma \Leftarrow \rho\rangle v \wedge \rho \not\lesssim \tau \\ \textit{BadCast}\, e &\equiv \exists E\ e'.\ e = E[e'] \wedge \textit{badcast}\, e' \end{aligned}$$

The (Merge) rule collapses two casts into a single cast, and is guarded by a type check. The target type of the resulting cast must be consistent with the inner source type $\rho$ and it must be a subtype of the outer target type $\tau$. We

**Fig. 3.** Reduction

$$\boxed{e \longrightarrow e}$$

(Ivk)
$$o.l_j(v) \longrightarrow [\mathsf{self} \mapsto o, x_j \mapsto v]e_j$$
$$(\text{where } o \equiv [l_i{=}\tau_i\ \varsigma(x_i\!:\!\sigma_i)e_i\ ^{i \in 1 \ldots n}]) \quad (1 \leq j \leq n)$$

(Delta)
$$c.l(v) \longrightarrow \delta(c, l, v)$$

(Upd)
$$[l_i{=}\tau_i\ \varsigma(x_i\!:\!\sigma_i)e_i\ ^{i \in 1 \ldots n}].l_j{:=}\tau\ \varsigma(x\!:\!\sigma)e$$
$$\longrightarrow [l_i{=}\tau_i\ \varsigma(x_i\!:\!\sigma_i)e_i\ ^{i \in \{1 \ldots n\} - \{j\}}, l_j{=}\tau\ \varsigma(x\!:\!\sigma)e] \quad (1 \leq j \leq n)$$

(Merge)
$$\frac{\rho \lesssim \tau \qquad \rho \neq \tau}{\langle \tau \Leftarrow \sigma' \rangle \langle \sigma \Leftarrow \rho \rangle v \longrightarrow \langle\!\langle (\rho \leftharpoondown \tau) \Leftarrow \rho \rangle\!\rangle v}$$

(Remove)
$$\frac{\rho = \tau}{\langle \tau \Leftarrow \sigma' \rangle \langle \sigma \Leftarrow \rho \rangle v \longrightarrow v}$$

(IvkCst)
$$(\langle \tau \Leftarrow \sigma \rangle v_1).l_j(v_2) \longrightarrow \langle \tau_2 \Leftarrow \sigma_2 \rangle (v_1.l_j(\langle \sigma_1 \Leftarrow \tau_1 \rangle v_2))$$
$$(\text{where } \sigma \equiv [\ldots, l_j : \sigma_1 \to \sigma_2, \ldots] \text{ and } \tau \equiv [\ldots, l_j : \tau_1 \to \tau_2, \ldots])$$

(UpdCst)
$$(\langle \tau \Leftarrow \sigma \rangle v).l_j :={\tau_2}\ \varsigma(x\!:\!\tau_1)e$$
$$\longrightarrow \langle \tau \Leftarrow \sigma \rangle (v.l_j :={\sigma_2}\ \varsigma(z\!:\!\sigma_1) \langle\!\langle \sigma_2 \Leftarrow \tau_2 \rangle\!\rangle [x \mapsto \langle\!\langle \tau_1 \Leftarrow \sigma_1 \rangle\!\rangle z]e)$$
$$(\text{where } \sigma \equiv [\ldots, l_j : \sigma_1 \to \sigma_2, \ldots] \text{ and } \tau \equiv [\ldots, l_j : \tau_1 \to \tau_2, \ldots])$$

$$\boxed{e \longmapsto e}$$

(Step)
$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}$$

$$\boxed{e \longmapsto^* e}$$

(Refl)
$$e \longmapsto^* e$$

(Trans)
$$\frac{e_1 \longmapsto^* e_2 \qquad e_2 \longmapsto e_3}{e_1 \longmapsto^* e_3}$$

therefore use the $\leftharpoondown$ operator and cast from $\rho$ to $\rho \leftharpoondown \tau$. The (Remove) rule applies when the inner source and the outer target types are equal, and removes both casts.

The delayed action of casts on objects is "forced" when a method is invoked or updated. The rules (IvkCst) and (UpdCst) handle these cases.

## 7 Type Safety of $\mathbf{Ob}^{?}_{<:}$

The bulk of this section is dedicated to proving that the intermediate language $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$ is type safe. The type safety of our source language $\mathbf{Ob}^{?}_{<:}$ is a consequence of the soundness of cast insertion and the type safety of the intermediate language. The type safety proof for the intermediate language has its origins in the syntactic type soundness approach of Wright and Felleisen[53], but is substantially reorganized using some folklore.[7] We begin with a top-down overview of the proof and then list the lemmas and theorems in the standard bottom-up fashion.

The goal is to show that if a term $e_s$ is well-typed ($\vdash e_s : \tau$) and reduces in zero or more steps to $e_f$ ($e_s \longmapsto^* e_f$), then $\vdash e_f : \tau$ and $e_f$ is either a value or contains a bad cast or $e_f$ can be further reduced. Note that the statement "$e_f$ is either a value or contains a bad cast or $e_f$ can be further reduced" is equivalent to saying that $e_f$ is not a *type error* as defined in Section 3. The proof of type safety is by induction on the reduction sequence. A reduction sequence (defined in Fig. 3) is either a zero-length sequence (so $e_s = e_f$), or a reduction sequence $e_s \longmapsto^* e_i$ to an intermediate term $e_i$ followed by a reduction step $e_i \longmapsto e_f$. In the zero-length case, where $e_s = e_f$, we need to show that if $e_s$ is well-typed then it is not a type error. This is shown in the Progress Lemma. In the second case, the induction hypothesis tells us that $e_i$ is well-typed. We then need to show that if $e_i$ is well-typed and $e_i \longmapsto e_f$ then $e_f$ is well-typed. This is shown in the Preservation Lemma. Once we have a well-typed $e_f$, we can use the Progress Lemma to show that $e_f$ is not a type error.

*Progress Lemma* Suppose that $e$ is well-typed and not a value and does not contain a bad cast. We need to show that $e$ can make progress, i.e., there is some $e'$ such that $e \longmapsto e'$. Therefore we need to show that $e$ can be decomposed into an evaluation context $E$ filled with a redex $e_1$ ($\exists e_2.\ e_1 \longrightarrow e_2$) so that we can apply rule (Step) to get $E[e_1] \longmapsto E[e_2]$. The existence of such a decomposition is given by the Decomposition Lemma.[8] In general, when the Progress Lemma fails for some language, it is because there is a mistake in the definition of evaluation contexts (which defines where evaluation should take place) or there is a mistake in the reduction rules, perhaps because a reduction rule is missing.

*Preservation Lemma* We need to show that if $\vdash e : \tau$ and $e \longmapsto e'$ then $\vdash e' : \tau$. Because $e \longmapsto e'$, we know there exists an $E$, $e_1$, and $e_2$ such that $e = E[e_1]$, $e' = E[e_2]$, and $e_1 \longrightarrow e_2$. The proof consists of three parts, each of which is proved as a separate lemma.

1. From $\vdash E[e_1] : \tau$ we know that $e_1$ is well-typed ($\vdash e_1 : \sigma$) and the context $E$ is well-typed. The typing judgment for contexts (defined the Appendix,

[7] The original proof of Wright and Felleisen requires the definition of faulty expressions which is more complicated than necessary because it relies on a proof by contradiction. Later type soundness proofs, such as [28, 38, 43], take a more direct approach. We use a proof organization similar to [5].

[8] Our Decomposition Lemma differs from the usual Unique Decomposition Lemma (but is similar to Lemma A.15 in [5]) in that we include the premise that the expression is well-typed and conclude with a stronger statement than usual, that the hole is filled with a redex. The usual approach is to conclude with a hole filled with something, let us call it a *pre-redex*, that turns out to be either a redex or an ill-typed term. We do not prove uniqueness here because it is not necessary in the proof of type safety. Nevertheless, decompositions are unique for $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$.

Fig. 5) assigns the context an input and output type, such as $\vdash E : \sigma \Rightarrow \tau$. (Subterm Typing)

2. Because $e_1$ is well-typed and $e_1 \longrightarrow e_2$, $e_2$ is well-typed with the same type as $e_1$. (Subject Reduction)
3. Filling $E$ with $e_2$ produces an expression of type $\tau$. More precisely, if $\vdash E : \sigma \Rightarrow \tau$ and $\vdash e_2 : \sigma$ then $\vdash E[e_2] : \tau$. (Replacement)

In general, Subterm Typing and Replacement hold for a language so long as evaluation contexts are properly defined. Subject Reduction, on the other hand, is highly dependent on the reduction rules of the language and is the crux of the type safety proof.

We now state the lemmas and theorems in the traditional bottom-up order, but without further commentary due to lack of space. We start with some basic properties of objects.

**Proposition 5 (Properties of Objects).**

1. *If* $\mathit{\Gamma} \vdash [l_i{=}\tau_i\ \varsigma(x_i{:}\sigma_i)e_i\ ^{i\in 1\ldots n}] : \rho$ *where* $\rho \equiv [l_i : \sigma_i \to \tau_i\ ^{i\in 1\ldots n}]$ *and* $j \in 1\ldots n$ *and* $\mathit{\Gamma}, \mathsf{self} : \rho, x_j : \sigma_j \vdash e' : \tau_j$ *then* $\mathit{\Gamma} \vdash [l_i{=}\tau_i\ \varsigma(x_i{:}\sigma_i)e_i\ ^{i\in\{1\ldots n\}-\{j\}}, l_j{=}\tau_j\ \varsigma(x_j : \sigma_j)e'] : \rho$.
2. *If* $[l_i : \sigma_i \to \tau_i\ ^{i\in 1\ldots n}] <: [l_j : \rho_j \to \nu_j\ ^{j\in 1\ldots m}]$ *and* $k \in 1\ldots m$ *then* $\rho_k = \sigma_k$ *and* $\nu_k = \tau_k$.

### 7.1 Progress

Towards proving the Progress Lemma, we show that values of certain types have canonical forms.

**Lemma 3 (Canonical Forms).**

1. *If* $\vdash v : \gamma$ *then* $\exists c \in \mathbb{C}.\ v = c$.
2. *If* $\vdash v : \rho$ *where* $\rho \equiv [l_i : \sigma_i \to \tau_i\ ^{i\in 1\ldots n}]$ *then* $\exists \overline{x}\ \overline{e}.\ v = [l_i = \tau_i\ \varsigma(x_i{:}\sigma_i)e_i\ ^{i\in 1\ldots n}]$ *or* $\exists \overline{x}\ \overline{e}\ \sigma.\ v = \langle \sigma \Leftarrow \rho\rangle[l_i = \tau_i\ \varsigma(x_i{:}\sigma_i)e_i\ ^{i\in 1\ldots n}]$.
3. $\not\vdash \xi : ?$ *(simple values do not have type ?)*

The main work in proving Progress is proving the Decomposition Lemma.

**Lemma 4 (Decomposition).** *If* $\vdash e : \tau$ *then* $e \in$ *Values or* $\exists \sigma\ E\ e'.\ e = E[e'] \wedge (\mathit{redex}\ e' \vee \mathit{badcast}\ e')$.

*Proof.* By induction on the typing derivation using the Canonical Forms Lemma and Proposition 5. □

**Lemma 5 (Progress).** *If* $\vdash e : \tau$ *then* $e \in$ *Values or* $\exists e'.e \longmapsto e'$ *or BadCast* $e$.

*Proof.* Immediate from the Decomposition Lemma. □

### 7.2 Preservation

Next we prove the Preservation Lemma and the three lemmas on which it relies: Subterm Typing, Subject Reduction, and Replacement.

**Lemma 6 (Subterm Typing).** *If* $\vdash E[e] : \tau$ *then* $\exists\sigma.\ \vdash E : \sigma \Rightarrow \tau$ *and* $\vdash e : \sigma$.

*Proof.* A straightforward induction on the typing derivation. □

We assume that the $\delta$ function for evaluating primitives is sound.

**Assumption 1** ($\delta$-typability)**.**
*If* $\mathit{TypeOf}(c) = [\ldots, l : \sigma \to \tau, \ldots]$ *and* $\vdash v : \sigma$ *then* $\vdash \delta(c, l, v) : \tau$.

Towards proving the Subject Reduction lemma, for the function application case we need the standard Substitution Lemma which in turn requires an Environment Weakening Lemma.

**Definition 7.** $\Gamma \subseteq \Gamma' \equiv \forall x\tau.\ \Gamma(x) = \tau$ *implies* $\Gamma'(x) = \tau$

**Lemma 7 (Environment Weakening).**
*If* $\Gamma \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash e : \tau$.

*Proof.* A straightforward induction on the typing derivation. □

**Definition 8.** *We write* $\Gamma\backslash\{x\}$ *for* $\Gamma$ *restricted to have domain* $\mathrm{dom}(\Gamma)\backslash\{x\}$.

**Lemma 8 (Substitution).**
*If* $\Gamma \vdash e_1 : \tau$ *and* $\Gamma(x) = \sigma$ *and* $\Gamma\backslash\{x\} \subseteq \Gamma'$ *and* $\Gamma' \vdash e_2 : \sigma$
*then* $\Gamma' \vdash [x \mapsto e_2]e_1 : \tau$.

*Proof.* By induction on the typing derivation. All cases are straightforward except for (Obj) and (Upd) for which we use Environment Weakening. □

**Lemma 9 (Inversions on Typing Rules).**

1. *If* $\Gamma \vdash c : \sigma \to \tau$ *then there exists* $\sigma'$ *and* $\tau'$ *such that* $\mathit{TypeOf}(c) = \sigma' \to \tau'$ *and* $\sigma <: \sigma'$ *and* $\tau' <: \tau$.
2. *If* $\Gamma \vdash \langle \tau' \Leftarrow \sigma \rangle e : \tau$ *then* $\tau' <: \tau$ *and* $\sigma \sim \tau'$ *and* $\sigma \neq \tau'$ *and* $\Gamma \vdash e : \sigma$.
3. *Suppose* $\Gamma \vdash [l_i = \tau_i\ \varsigma(x_i : \sigma_i)e_i\ ^{i \in 1 \ldots n}] : \tau$ *and let* $\rho \equiv [l_i : \sigma_i \to \tau_i\ ^{i \in 1 \ldots n}]$. *Then* $\rho <: \tau$ *and for any* $j \in 1 \ldots n$ *we have* $\Gamma, \mathsf{self} : \rho, x_j : \sigma_j \vdash e_j : \tau_j$.

*Proof.* The proofs are by induction on the typing derivation. □

**Lemma 10 (Subject Reduction).** *If* $\vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\vdash e' : \tau$.

*Proof.* The proof is by induction on the typing derivation, followed by case analysis on the reduction.

**(Ivk)** Use the Substitution and Inversion Lemmas and Proposition 5.
**(Delta)** Use $\delta$-typability and the Inversion Lemma.
**(Upd)** Use Proposition 5 and the Inversion Lemma.

**(Merge)** Use Proposition 3 and the Inversion Lemma.
**(Remove, InvCst, UpdCst)** Use the Inversion Lemma.

□

**Lemma 11 (Replacement).** *If* $E : \sigma \Rightarrow \tau$ *and* $\vdash e : \sigma$ *then* $\vdash E[e] : \tau$.

*Proof.* A straightforward induction on the context typing derivation. □

**Lemma 12 (Preservation).** *If* $e \longmapsto e'$ *and* $\vdash e : \tau$ *then* $\vdash e' : \tau$.

*Proof.* Apply Subterm Typing to get a well-typed evaluation context and redex. Then apply Subject Reduction and Replacement. □

### 7.3 Type Safety

**Lemma 13 (Type Safety of $\mathbf{Ob}^{\langle\cdot\rangle}_{<:}$).** *If* $\vdash e : \tau$ *and* $e \longmapsto^* e'$ *then* $\vdash e' : \tau$ *and* $e' \in$ *Values or BadCast* $e'$ *or* $\exists e''.\ e' \longmapsto e''$.

*Proof.* By induction on the evaluation steps. For the base case, where $e = e'$, we use Progress to show that $e$ is either a value, a bad cast, or can make progress. For the case where $e_1 \longmapsto^* e_2$ and $e_2 \longmapsto e_3$, $e_2$ is well-typed by the induction hypothesis and therefore $e_3$ is well-typed by Preservation. Applying Progress to $e_3$ brings us to the conclusion. □

**Theorem 1 (Type Safety of $\mathbf{Ob}^{?}_{<:}$).** *If* $\vdash e_1 \rightsquigarrow e_2 : \tau$ *and* $e_2 \longmapsto^* e_3$ *then* $\vdash e_3 : \tau$ *and* $e_3 \in$ *Values or BadCast* $e_3$ *or* $\exists e_4.\ e_3 \longmapsto e_4$.

*Proof.* The expression $e_2$ is well-typed because cast insertion is sound (Lemma 1). We then apply Lemma 13. □

**Theorem 2 (Static Type Safety of $\mathbf{Ob}^{?}_{<:}$).** *If* $e_1 \in \mathbf{Ob}_{<:}$ *and* $\vdash e_1 \rightsquigarrow e_2 : \tau$ *and* $e_2 \longmapsto^* e_3$ *then* $\vdash e_3 : \tau$ *and* $e_3 \in$ *Values or* $\exists e_4.\ e_3 \longmapsto e_4$.

*Proof.* By Lemma 2 we have $e_1 = e_2$, so $e_2$ does not contain any casts. By Lemma 13 we know that either $e_3$ is a value or a bad cast or can make progress. However, since $e_2$ did not contain any casts, there can be none in $e_3$. □

## 8 Related Work

*Type Annotations for Dynamic Languages* Several dynamic programming languages allow explicit type annotations, such as Common LISP [33], Dylan [16, 45], Cecil [10], Boo [13], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [36], the Bigloo [8, 44] dialect of Scheme [34], and the Strongtalk dialect of Smalltalk [6, 7]. 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. This paper formalizes a type system that provides this stronger guarantee.

*Soft Typing* Static checking can be added to dynamically typed languages using static analyses. Cartwright and Fagan [9], Flanagan and Felleisen [22], Aiken, Wimmers, and Lakshman [3], and Henglein and Rehof [29, 30] developed analyses that can be used, for example, to catch bugs in Scheme programs [23, 30]. 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. Also, the static analyses bear a significant implementation burden on developers of the language. On the other hand, they can be used to reduce the amount of run-time type checking in dynamically typed programs (Chambers et al. [11, 14]) and therefore could also be used to improve the performance of gradually typed programs.

*Dynamic Typing in Statically Typed Languages* Abadi et al. [2] extended a statically typed language with a Dynamic type and explicit injection (dynamic) and projection operations (typecase). Their approach does not satisfy our goals, 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. Our approach automates the latter.

*Interoperability* Gray, Findler, and Flatt [25] 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 [35] 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] 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 errors may not originate from statically typed modules. Our gradual type system enjoys a similar property. If all parameters in a term are annotated then no casts are inserted into the term during compilation provided the types of the free variables in the term do not mention ? (Lemma 2). Thus, no cast errors can originate from such a term.

*Hybrid typing* The Hybrid Type Checking of Flanagan et al. [21, 24] combines standard static typing with refinement types, where the refinements may express arbitrary predicates. The type system tries to satisfy the predicates using automated theorem proving, but when no conclusive answer is given, the system inserts run-time checks. This work is analogous to ours in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. One notable difference between our system and Flanagan's is that his is based on subtyping whereas ours is based on type consistency.

Ou et al. [40] 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.

*Quasi-Static Typing* Thatte's Quasi-Static Typing [50] is close to our gradual type system but relies on subtyping and treats the unknown type as the top of the subtype hierarchy. In previous work [47] we showed that implicit down-casts combined with the transitivity of subtyping creates a fundamental problem that prevents the 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 D$\pi$, a distributed $\pi$-calculus. Their system allows some locations to be untyped and assigns such locations the type $\mathsf{lbad}$. Their type system, like Quasi-Static Typing, relies on subtyping, however they treat $\mathsf{lbad}$ as "bottom", which allows objects of type $\mathsf{lbad}$ to be implicitly coercible to any other type.

*Gradual Typing* The work of Anderson and Drossopoulou on BabyJ [4] is closest to our own. They develop a gradual type system for *nominal types* and their permissive type $*$ is analogous to our unknown type ?. Our work differs from theirs in that we address structural type systems.

Gronski, Knowles, Tomb, Freund, and Flanagan [26] provide gradual typing in the Sage language by including a $\mathsf{Dynamic}$ type and implicit down-casts. They use a modified form of subtyping to provide the implicit down-casts whereas we use the consistency relation. Their work does not include a result such as Theorem 2 of this paper which shows that all type errors are caught in programs with fully annotated parameters.

Herman alerted us to the space-efficiency problems in the traditional approach to higher-order casts. (We used the traditional approach in [47].) Concurrent to the work in this paper, Herman, Tomb, and Flanagan [31] proposed a solution a space-efficiency problem which, similar to our approach, delays the application of higher-order casts. However, the details of their approach are based on the coercion calculus from Henglein's Dynamic Typing framework [29]. The coercion calculus can be viewed as a way to *compile* the explicit casts of this paper, removing the interpretive overhead of traversing types at run-time.

*Type inference* A language with gradual typing is syntactically similar to one with type inference [12, 32, 37]: both allow type annotations to be omitted. However, type inference does not provide the same benefits as dynamic typing (and therefore gradual typing). With type inference, programmers save the time it takes to write down the types but they must still go through the process of revising their program until the type inferencer accepts the program as well typed. As type systems are conservative in nature and of limited (though ever increasing) expressiveness, it may take some time to turn a program (even one without any real errors) into a program to which the type inferencer can assign a type. The advantage of dynamic typing (and therefore of gradual typing) is that programmers may begin executing and testing their programs right away.

## 9 Conclusion and Future Work

The debate between dynamic and static typing has continued for several decades, with good reason. There are convincing arguments for both sides. Dynamic typing is better suited for prototyping, scripting, and gluing components, whereas static typing is better suited for algorithms, data-structures, and systems programming. It is common practice for programmers to start developing a program in a dynamic language and then translate to a static language later on. However, static and dynamic languages are often radically different, making this translation difficult and error prone. Ideally, migrating between dynamic to static could take place gradually and within one language.

In this paper we present the formal definition of an object calculus $\mathbf{Ob}^{?}_{<:}$, including its type system and operational semantics. This language captures the key ingredients for implementing gradual typing in object-oriented languages, showing how the type consistency relation can be naturally combined with subtyping. The calculus $\mathbf{Ob}^{?}_{<:}$ provides the flexibility of dynamically typed languages when type annotations are omitted by the programmer and provides the benefits of static checking when all method parameters are annotated. The type system and run-time semantics of $\mathbf{Ob}^{?}_{<:}$ are relatively straightforward, so it is suitable for practical languages.

As future work, we intend to investigate the interaction between gradual typing and Hindley-Milner inference [12, 32, 37], and we intend to apply static analyses (such as Soft Typing [9] or Henglein's Dynamic Typing [29]) to reduce the number of run-time casts that must be inserted during compilation. There are a number of features we omitted from the formalization for the sake of keeping the presentation simple, such as recursive types and imperative update. We plan to add these features to our formalization in the near future. Finally, we intend to incorporate gradual typing into a mainstream dynamically typed programming language and perform studies to evaluate whether gradual typing can benefit programmer productivity.

## Acknowledgments

We thank the anonymous reviewers for their suggestions. We thank Amer Diwan, Christoph Reichenbach, Ronald Garcia, Stephen Freund, David Herman, David Broman, and Cormac Flanagan for comments and feedback on drafts of this paper. This work was supported by NSF ITR-0113569 Putting Multi-Stage Annotations to Work, Texas ATP 003604-0032-2003 Advanced Languages Techniques for Device Drivers, and NSF SOD-0439017 Synthesizing Device Drivers.

# Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] 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.

[3] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

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

[5] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. W. S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. May 2005.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 49–70, New York, NY, USA, 1989. ACM Press.

[12] 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.

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

[14] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102, New York, NY, USA, 1995. ACM Press.

[15] ECMA. *Standard ECMA-262: ECMAScript Language Specification*, 1999.

[16] 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.

[17] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the lambda-calculus. pages 193–217, 1986.

[18] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[19] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.

[20] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.

[21] 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.

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

[23] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 23–32, New York, NY, USA, 1996. ACM Press.

[24] 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.

[25] 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.

[26] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. Technical report, University of California, Santa Cruz, 2006.

[27] E. T. W. Group. Ecmascript 4 netscape proposal.

[28] C. A. Gunter, D. Remy, and J. G. Riecke. A generalization of exceptions and control in ml-like languages. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, New York, NY, USA, 1995. ACM Press.

[29] F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[30] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. In *FPCA '95, ACM SIGPLAN-SIGARCH Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] A. Nanevski. A modal calculus for exception handling. In *Intuitionistic Modal Logics and Applications Workshop (IMLA '05)*, Chicago, IL, June 2005.

[39] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[40] 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.

[41] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[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] A. Sabry. Minml: Syntax, static semantics, dynamic semantics, and type safety. Course notes for b522, February 2002.

[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 W. Taha. Gradual typing for objects: Isabelle formaliztaion. Technical Report CU-CS-1021-06, University of Colorado, Boulder, CO, December 2006.

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

[48] W. Taha, H. Makholm, and J. Hughes. Tag elimination and jones-optimality. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 257–275, London, UK, 2001. Springer-Verlag.

[49] A. Tang. Pugs blog.

[50] S. Thatte. Quasi-static typing. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381, New York, NY, USA, 1990. ACM Press.

[51] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

[52] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, April 2004.

[53] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

# Appendix

**Fig. 4.** The type system for $\mathbf{Ob}_{<:}$.

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

$$(\textsc{Var}) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$(\textsc{Const}) \qquad \Gamma \vdash c : \mathit{TypeOf}(c)$$

$$(\textsc{Obj}) \qquad \frac{\Gamma, \mathsf{self} : \rho, x_i : \sigma_i \vdash e_i : \tau_i \qquad \forall i \in 1 \ldots n}{\Gamma \vdash [l_i = \tau_i \; \varsigma(x_i : \sigma_i) e_i \; ^{i \in 1 \ldots n}] : \rho} \quad \text{(where } \rho \equiv [l_i : \sigma_i \to \tau_i \; ^{i \in 1 \ldots n}]\text{)}$$

$$(\textsc{Ivk}) \qquad \frac{\Gamma \vdash e_1 : [\ldots, l : \sigma \to \tau, \ldots] \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1.l(e_2) : \tau}$$

$$(\textsc{Upd}) \qquad \frac{\Gamma \vdash e_1 : \rho \qquad \Gamma, \mathsf{self} : \rho, x : \sigma \vdash e_2 : \tau \qquad \sigma_k <: \sigma \qquad \tau <: \tau_k}{\Gamma \vdash e_1.l_k := \tau \; \varsigma(x : \sigma) e_2 : \rho} \quad \text{(where } \rho \equiv [l_i : \sigma_i \to \tau_i \; ^{i \in 1 \ldots n}] \text{ and } k \in 1 \ldots n\text{)}$$

$$(\textsc{Sub}) \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

**Fig. 5.** Well-typed contexts.

$$\boxed{\vdash E : \tau \Rightarrow \tau}$$

$$(\textsc{CxHole}) \qquad \vdash [] : \tau \Rightarrow \tau$$

$$(\textsc{CxIvkL}) \qquad \frac{\vdash E : \sigma \Rightarrow [\ldots, l : \rho \to \tau, \ldots] \qquad \vdash e : \rho}{\vdash E.l(e) : \sigma \Rightarrow \tau}$$

$$(\textsc{CxIvkR}) \qquad \frac{\vdash e : [\ldots, l : \rho \to \tau, \ldots] \qquad \vdash E : \sigma \Rightarrow \rho}{\vdash e.l(E) : \sigma \Rightarrow \tau}$$

$$(\textsc{CxUpd}) \qquad \frac{\vdash E : \sigma' \Rightarrow \rho \qquad \mathsf{self} : \rho, x : \sigma \vdash e : \tau \qquad \sigma_k <: \sigma \qquad \tau <: \tau_k}{\vdash E.l_k := \tau \; \varsigma(x : \sigma) e : \sigma' \Rightarrow \rho} \quad \text{(where } \rho \equiv [l_i : \sigma_i \to \tau_i \; ^{i \in 1 \ldots n}] \text{ and } 1 \le k \le n\text{)}$$

$$(\textsc{CxSub}) \qquad \frac{\vdash E : \sigma \Rightarrow \rho \qquad \vdash \rho <: \rho'}{\vdash E : \sigma \Rightarrow \rho'}$$