Gradual Typing for Functional Languages

Jeremy G. Siek
University of Colorado
siek@cs.colorado.edu

Walid Taha
Rice University
taha@rice.edu

Abstract

Static and dynamic type systems have well-known strengths and weaknesses, and each is better suited for different programming tasks. There have been many efforts to integrate static and dynamic typing and thereby combine the benefits of both typing disciplines in the same language. The flexibility of static typing can be improved by adding a type Dynamic and a typecase form. The safety and performance of dynamic typing can be improved by adding optional type annotations or by performing type inference (as in soft typing). However, there has been little formal work on type systems that allow a programmer-controlled migration between dynamic and static typing. Thatte proposed Quasi-Static Typing, but it does not statically catch all type errors in completely annotated programs. Anderson and Drossopoulou defined a nominal type system for an object-oriented language with optional type annotations. However, developing a sound, gradual type system for functional languages with structural types is an open problem.

In this paper we present a solution based on the intuition that the structure of a type may be partially known/unknown at compile-time and the job of the type system is to catch incompatibilities between the known parts of types. We define the static and dynamic semantics of a λ-calculus with optional type annotations and we prove that its type system is sound with respect to the simply-typed λ-calculus for fully-annotated terms. We prove that this calculus is type safe and that the cost of dynamism is "pay-as-you-go".

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs— *Type structure*

General Terms Languages, Performance, Theory

Keywords static and dynamic typing, optional type annotations

1. Introduction

Static and dynamic typing have different strengths, making them better suited for different tasks. Static typing provides early error detection, more efficient program execution, and better documentation, whereas dynamic typing enables rapid development and fast adaptation to changing requirements.

The focus of this paper is languages that literally provide static and dynamic typing in the same program, with the programmer controlling the degree of static checking by annotating function parameters with types, or not. We use the term ***gradual typing*** for type systems that provide this capability. Languages that support gradual typing to a large degree include Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], and extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. The purpose of this paper is to provide a type-theoretic foundation for languages such as these with gradual typing.

There are numerous other ways to combine static and dynamic typing that fall outside the scope of gradual typing. Many dynamically typed languages have optional type annotations that are used to improve run-time performance but not to increase the amount of static checking. Common LISP [23] and Dylan [12, 37] are examples of such languages. Similarly, the Soft Typing of Cartwright and Fagan [7] improves the performance of dynamically typed languages but it does not statically catch type errors. At the other end of the spectrum, statically typed languages can be made more flexible by adding a Dynamic type and typecase form, as in the work by Abadi et al. [1]. However, such languages do not allow for programming in a dynamically typed style because the programmer is required to insert coercions to and from type Dynamic.

A short example serves to demonstrate the idea of gradual typing. Figure 1 shows a call-by-value interpreter for an applied λ-calculus written in Scheme extended with gradual typing and algebraic data types. The version on the left does not have type annotations, and so the type system performs little type checking and instead many tag-tests occur at run time.

As development progresses, the programmer adds type annotations to the parameters of interp, as shown on the right side of Figure 1, and the type system provides more aid in detecting errors. We use the notation ? for the dynamic type. The type system checks that the uses of env and e are appropriate: the case analysis on e is fine and so is the application of assq to x and env. The recursive calls to interp also type check and the call to **apply** type checks trivially because the parameters of **apply** are dynamic. Note that we are still using dynamic typing for the value domain of the object language. To obtain a program with complete static checking, we would introduce a datatype for the value domain and use that as the return type of interp.

Contributions We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated. These benefits include both safety and performance: type errors are caught at compile-time and values may be stored in unboxed form. That is, for statically typed portions of the program there is no need for run-time tags and tag checking.

We introduce a calculus named $\lambda^{?}_{\rightarrow}$ and define its type system (Section 2). We show that this type system, when applied to fully an-

Scheme and Functional Programming 2006 September 17, Portland, OR.

```
(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

```
(type expr (datatype (Var ,symbol)
                     (Int ,int)
                     (App ,expr ,expr)
                     (Lam ,symbol ,expr)
                     (Succ ,expr)))
(type envty (listof (pair symbol ?)))

(define interp
  (λ ((env : envty) (e : expr))
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

Figure 1. An example of gradual typing: an interpreter with varying amounts of type annotations.

notated terms, is equivalent to that of the simply-typed lambda calculus (Theorem 1). This property ensures that for fully-annotated programs all type errors are caught at compile-time. Our type system is the first gradual type system for structural types to have this property. To show that our approach to gradual typing is suitable for imperative languages, we extend $\lambda^{?}_{\rightarrow}$ with ML-style references and assignment (Section 4).

We define the run-time semantics of $\lambda^{?}_{\rightarrow}$ via a translation to a simply-typed calculus with explicit casts, $\lambda^{\langle\tau\rangle}_{\rightarrow}$, for which we define a call-by-value operational semantics (Section 5). When applied to fully-annotated terms, the translation does not insert casts (Lemma 4), so the semantics exactly matches that of the simply-typed λ-calculus. The translation preserves typing (Lemma 3) and $\lambda^{\langle\tau\rangle}_{\rightarrow}$ is type safe (Lemma 8), and therefore $\lambda^{?}_{\rightarrow}$ is type safe: if evaluation terminates, the result is either a value of the expected type or a cast error, but never a type error (Theorem 2).

On the way to proving type safety, we prove Lemma 5 (Canonical Forms), which is of particular interest because it shows that the run-time cost of dynamism in $\lambda^{?}_{\rightarrow}$ can "pay-as-you-go". Run-time polymorphism is restricted to values of type ?, so for example, a value of type int must actually be an integer, whereas a value of type ? may contain an integer or a Boolean or anything at all. Compilers for $\lambda^{?}_{\rightarrow}$ may use efficient, unboxed, representations for values of ground and function type, achieving the performance benefits of static typing for the parts of programs that are statically typed.

The proofs of the lemmas and theorems in this paper were written in the Isar proof language [28, 42] and verified by the Isabelle proof assistant [29]. We provide proof sketches in this paper and the full proofs are available in the companion technical report [39]. The statements of the definitions (including type systems and semantics), lemmas, propositions, and theorems in this paper were automatically generated from the Isabelle files. Free variables that appear in these statements are universally quantified.

2. Introduction to Gradual Typing

The gradually-typed λ-calculus, $\lambda^{?}_{\rightarrow}$, is the simply-typed λ-calculus extended with a type ? to represent dynamic types. We present gradual typing in the setting of the simply-typed λ-calculus to reduce unnecessary distractions. However, we intend to show how gradual typing interacts with other common language features, and as a first step combine gradual typing with ML-style references in Section 4.

Syntax of the Gradually-Typed Lambda Calculus $e \in \lambda^{?}_{\rightarrow}$

Variables	$x \in \mathbb{X}$		
Ground Types	$\gamma \in \mathbb{G}$		
Constants	$c \in \mathbb{C}$		
Types	τ	$::=$	$\gamma \mid ? \mid \tau \rightarrow \tau$
Expressions	e	$::=$	$c \mid x \mid \lambda x\!:\!\tau.\ e \mid e\ e$
			$\lambda x.\ e \equiv \lambda x\!:\!?.\ e$

A procedure without a parameter type annotation is syntactic sugar for a procedure with parameter type ?.

The main idea of our approach is the notion of a type whose structure may be partially known and partially unknown. The unknown portions of a type are indicated by ?. So, for example, the type **number** $*$? is the type of a pair whose first element is of type **number** and whose second element has an unknown type. To program in a dynamically typed style, omit type annotations on parameters; they are by default assigned the type ?. To enlist more help from the type checker, add type annotations, possibly with ? occurring inside the types to retain some flexibility.

The job of the static type system is to reject programs that have inconsistencies in the known parts of types. For example, the program

```
((λ (x : number) (succ x)) #t) ;; reject
```

should be rejected because the type of #t is not consistent with the type of the parameter x, that is, **boolean** is not consistent with **number**. On the other hand, the program

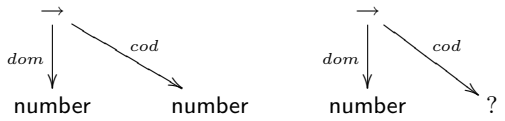
```
((λ (x) (succ x)) #t) ;; accept
```

should be accepted by the type system because the type of x is considered unknown (there is no type annotation) and therefore not within the realm of static checking. Instead, the type error will be caught at run-time (as is typical of dynamically typed languages), which we describe in Section 5.

As usual things become more interesting with first class procedures. Consider the following example of mapping a procedure over a list.

```
map : (number → number) * number list → number list
(map (λ (x) (succ x)) (list 1 2 3)) ;; accept
```

The **map** procedure is expecting a first argument whose type is **number** → **number** but the argument (λ(x) (succ x)) has type ? → **number**. We would like the type system to accept this program, so how should we define consistency for procedure types? The intuition is that we should require the known portions of the two types to be equal and ignore the unknown parts. There is a useful analogy with the mathematics of partial functions: two partial functions are ***consistent*** when every elements that is in the domain of both functions is mapped to the same result. This analogy can be made formal by considering types as trees [32].



Trees can be represented as partial functions from paths to node labels, where a path is a sequence of edge labels: $[l_1, \ldots, l_n]$. The above two trees are the following two partial functions f and g. We interpret unknown portions of a type simply as places where the partial function is undefined. So, for example, g is undefined for the path $[cod]$.

$$\begin{aligned} f([]) &= \rightarrow \\ f([dom]) &= number \\ f([cod]) &= number \end{aligned}$$

$$\begin{aligned} g([]) &= \rightarrow \\ g([dom]) &= number \end{aligned}$$

The partial functions f and g are consistent because they produce the same output for the inputs [] and $[dom]$.

We axiomatize the consistency relation $\sim$ on types with the following definition.

Type Consistency $\tau \sim \tau$

$$\text{(CRefl)}\ \tau \sim \tau \qquad \text{(CFun)}\ \frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{\sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2}$$

$$\text{(CUnR)}\ \tau \sim ? \qquad \text{(CUnL)}\ ? \sim \tau$$

The type consistency relation is reflexive and symmetric but not transitive (just like consistency of partial functions).

Proposition 1.

- $\tau \sim \tau$
- If $\sigma \sim \tau$ then $\tau \sim \sigma$.
- $\neg\,(\forall\, \tau_1\ \tau_2\ \tau_3.\ \tau_1 \sim \tau_2 \wedge \tau_2 \sim \tau_3 \longrightarrow \tau_1 \sim \tau_3)$

Our gradual type system is shown in Figure 2. The environment Γ is a function from variables to optional types ($\lfloor\tau\rfloor$ or $\bot$). The type system is parameterized on a signature Δ that assigns types to constants. The rules for variables, constants, and functions are standard. The first rule for function application (GApp1) handles the case when the function type is unknown. The argument may have any type and the resulting type of the application is unknown. The second rule for function application (GApp2) handles when the function type is known and allows an argument whose type is consistent with the function's parameter type.

Figure 2. A Gradual Type System

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

$$\text{(GVar)}\ \frac{\Gamma\, x = \lfloor\tau\rfloor}{\Gamma \vdash_G x : \tau} \qquad \text{(GConst)}\ \frac{\Delta\, c = \tau}{\Gamma \vdash_G c : \tau}$$

$$\text{(GLam)}\ \frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda\, x{:}\sigma.\ e : \sigma \rightarrow \tau}$$

$$\text{(GApp1)}\ \frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\ e_2 : ?}$$

$$\text{(GApp2)}\ \frac{\Gamma \vdash_G e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash_G e_1\ e_2 : \tau'}$$

Relation to the untyped λ-calculus We would like our gradual type system to accept all terms of the untyped λ-calculus (all unannotated terms), but it is not possible to simultaneously achieve this and provide type safety for fully-annotated terms. For example, suppose there is a constant succ with type **number** → **number**. The term (succ "hi") has no type annotations but it is also fully annotated because there are no function parameters to annotate. The type system must either accept or reject this program. We choose to reject. Of course, if succ were given the type ? → ? then (succ "hi") would be accepted. In any event, our gradual type system provides the same expressiveness as the untyped λ-calculus. The following translation converts any λ-term into an observationally equivalent well-typed term of $\lambda^{?}_{\rightarrow}$.

$$\begin{aligned} [\![c]\!] &= c \\ [\![x]\!] &= x \\ [\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\ [\![e_1\ e_2]\!] &= ((\lambda x.x)[\![e_1]\!])[\![e_2]\!] \end{aligned}$$

Relation to the simply-typed λ-calculus Let $\lambda_{\rightarrow}$ denote the terms of the simply-typed λ-calculus and let $\Gamma \vdash_{\rightarrow} e : \tau$ stand for the standard typing judgment of the simply-typed λ-calculus. For terms in $\lambda_{\rightarrow}$ our gradual type system is equivalent to simple typing.

Theorem 1 (Equivalence to simple typing for fully-annotated terms)**.** If $e \in \lambda_{\rightarrow}$ then $\emptyset \vdash_G e : \tau = \emptyset \vdash_{\rightarrow} e : \tau$.

Proof Sketch. The rules for our gradual type system are the same as for the STLC if one removes the rules that mention ?. The type compatibility relation collapses to type equality once all rules involving ? are removed. □

A direct consequence of this equivalence is that our gradual type system catches the same static errors as the type system for $\lambda_{\rightarrow}$.

Corollary 1 (Full static error detection for fully-annotated terms)**.** If $e \in \lambda_{\rightarrow}$ and $\nexists \tau.\ \emptyset \vdash_{\rightarrow} e : \tau$ then $\nexists \tau'.\ \emptyset \vdash_G e : \tau'$. *(This is just the contrapositive of soundness.)*

Before describing the run-time semantics of $\lambda^?_\to$ we compare our type system for $\lambda^?_\to$ with an alternative design based on subtyping.

3. Comparison with Quasi-Static Typing

Our first attempt to define a gradual type system was based on Thatte's quasi-static types [40]. Thatte uses a standard subtyping relation $<:$ with a top type Ω to represent the dynamic type. As before, the meta-variable γ ranges over ground types such as **number** and **boolean**.

Subtyping rules. $\tau <: \tau'$

$$\frac{}{\gamma <: \gamma} \qquad \frac{}{\tau <: \Omega} \qquad \frac{\sigma_1 <: \tau_1 \quad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}$$

The quasi-static type system includes the usual subsumption rule.

$$\text{QSUB}\ \frac{\Gamma \vdash e : \tau \quad \tau <: \sigma}{\Gamma \vdash e : \sigma}$$

Subsumption allows programs such as the following to type check by allowing implicit up-casts. The value #t of type **boolean** is up-cast to Ω, the type of the parameter x.

```
((λ (x) ...) #t) ;; ok, boolean <: Ω
```

However, the subsumption rule will not allow the following program to type check. The addition operator expects type **number** but gets an argument of type Ω.

```
(λ (x) (succ x))
```

Thatte's solution for this is to also allow an implicit down-cast in the (QAPP2) rule for function application.

$$(\text{QAPP2})\ \frac{\Gamma \vdash e_1 : \sigma \to \sigma' \quad \Gamma \vdash e_2 : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1\ e_2) : \sigma'}$$

Unfortunately, the subsumption rule combined with (QAPP2) allows too many programs to type check for our taste. For example, we can build a typing derivation for the following program, even though it was rejected by our gradual type system.

```
((λ (x : number) (succ x)) #t)
```

The subsumption rule allows #t to be implicitly cast to Ω and then the above rule for application implicitly casts Ω down to **number**.

To catch errors such as these, Thatte added a second phase to the type system called plausibility checking. This phase rewrites the program by collapsing sequences of up-casts and down-casts and signals an error if it encounters a pair of casts that together amount to a "stupid cast"[22], that is, casts that always fail because the target is incompatible with the subject.

Figure 3 shows Thatte's Quasi-Static type system. The judgment $\Gamma \vdash e \Rightarrow e' : \tau$ inserts up-casts and down-casts and the judgment $e \rightsquigarrow e'$ collapses sequences of casts and performs plausibility checking. The type system is parameterized on the function Δ mapping constants to types. The environment Γ is a function from variables to optional types ($\lfloor\tau\rfloor$ or $\bot$).

Subsumption rules are slippery, and even with the plausibility checks the type system fails to catch many errors. For example, there is still a derivation for the program

```
((λ (x : number) (succ x)) #t)
```

The reason is that both the operator and operand may be implicitly up-cast to Ω. The rule (QAPP1) then down-casts the operator to $\Omega \to \Omega$. Plausibility checking succeeds because there is a greatest lower bound of **number** $\to$ **number** and $\Omega \to \Omega$, which is $\Omega \to$ **number**. So the quasi-static system fails to statically catch the type error.

Figure 3. Thatte's Quasi-Static Typing.

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

$$(\text{QVAR})\ \frac{\Gamma\ x = \lfloor\tau\rfloor}{\Gamma \vdash x \Rightarrow x : \tau}$$

$$(\text{QCONST})\ \frac{\Delta\ c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$$

$$(\text{QLAM})\ \frac{\Gamma, x : \tau \vdash e \Rightarrow e' : \sigma}{\Gamma \vdash (\lambda x{:}\tau.\ e) \Rightarrow (\lambda x{:}\tau.\ e') : \tau \to \sigma}$$

$$(\text{QSUB})\ \frac{\Gamma \vdash e \Rightarrow e' : \tau \quad \tau <: \sigma}{\Gamma \vdash e \Rightarrow e' \uparrow^\sigma_\tau : \sigma}$$

$$(\text{QAPP1})\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \Omega \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau}{\Gamma \vdash (e_1\ e_2) \Rightarrow ((e_1' \downarrow^\Omega_{\tau\to\Omega})\ e_2') : \Omega}$$

$$(\text{QAPP2})\ \frac{\Gamma \vdash e_1 \Rightarrow e_1' : \sigma \to \sigma' \quad \Gamma \vdash e_2 \Rightarrow e_2' : \tau \quad \sigma <: \tau}{\Gamma \vdash (e_1\ e_2) \Rightarrow (e_1'\ (e_2' \downarrow^\tau_\sigma)) : \sigma'}$$

$\boxed{e \rightsquigarrow e'}$

$$e \downarrow^\tau_\tau \rightsquigarrow e \qquad e \uparrow^\tau_\tau \rightsquigarrow e$$

$$e \downarrow^\tau_\sigma \downarrow^\sigma_\mu \rightsquigarrow e \downarrow^\tau_\mu \qquad e \uparrow^\sigma_\mu \uparrow^\tau_\sigma \rightsquigarrow e \uparrow^\tau_\mu$$

$$\frac{\mu = \tau \sqcap \nu}{e \uparrow^\sigma_\tau \downarrow^\sigma_\nu \rightsquigarrow e \downarrow^\tau_\mu \uparrow^\nu_\mu} \qquad \frac{\nexists \mu.\mu = \tau \sqcap \nu}{e \uparrow^\sigma_\tau \downarrow^\sigma_\nu \rightsquigarrow \mathsf{wrong}}$$

As noted by Oliart [30], Thatte's quasi-static type system does not correspond to his type checking *algorithm* (Theorem 7 of [40] is incorrect). Thatte's type checking algorithm does not suffer from the above problems because the algorithm does not use the subsumption rule and instead performs all casting at the application rule, disallowing up-casts to Ω followed by arbitrary down-casts. Oliart defined a simple syntax-directed type system that is equivalent to Thatte's algorithm, but did not state or prove any of its properties. We initially set out to prove type safety for Oliart's subtype-based type system, but then realized that the consistency relation provides a much simpler characterization of when implicit casts should be allowed.

At first glance it may seem odd to use a symmetric relation such as consistency instead of an anti-symmetric relation such as subtyping. There is an anti-symmetric relation that is closely related to consistency, the usual partial ordering relation for partial functions: $f \sqsubseteq g$ if the graph of f is a subset of the graph of g. (Note that the direction is flipped from that of the subtyping relation $<:$, where greater means less information.) A cast from τ to σ, where $\sigma \sqsubseteq \tau$, always succeeds at run-time as we are just hiding type information by replacing parts of a type with ?. On the other hand, a cast from σ to τ may fail because the run-time type of the value may not be consistent with τ. The main difference between $\sqsubseteq$ and $<:$ is that $\sqsubseteq$ is covariant for the domain of a procedure type, whereas $<:$ is contra-variant for the domain of a procedure type.

Figure 4. Type Rules for References

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

$$\text{(GRef)} \quad \frac{\Gamma \vdash_G e : \tau}{\Gamma \vdash_G \mathit{ref}\ e : \mathit{ref}\ \tau}$$

$$\text{(GDeref1)} \quad \frac{\Gamma \vdash_G e : ?}{\Gamma \vdash_G !e : ?}$$

$$\text{(GDeref2)} \quad \frac{\Gamma \vdash_G e : \mathit{ref}\ \tau}{\Gamma \vdash_G !e : \tau}$$

$$\text{(GAssign1)} \quad \frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \mathit{ref}\ \tau}$$

$$\text{(GAssign2)} \quad \frac{\Gamma \vdash_G e_1 : \mathit{ref}\ \tau \qquad \Gamma \vdash_G e_2 : \sigma \qquad \sigma \sim \tau}{\Gamma \vdash_G e_1 \leftarrow e_2 : \mathit{ref}\ \tau}$$

4. Gradual Typing and References

It is often challenging to integrate type system extensions with imperative features such as references with assignment. In this section we extend the calculus to include ML-style references. The following grammar shows the additions to the syntax.

Adding references to $\lambda^{?}_{\rightarrow}$

Types	τ	::=	$\ldots \mid \mathit{ref}\ \tau$
Expressions	e	::=	$\ldots \mid \mathit{ref}\ e \mid !e \mid e \leftarrow e$

The form $\mathit{ref}\ \ e$ creates a reference cell and initializes it with the value that results from evaluating expression e. The dereference form $!e$ evaluates e to the address of a location in memory (hopefully) and returns the value stored there. The assignment form $e \leftarrow e$ stores the value form the right-hand side expression in the location given by the left-hand side expression.

Figure 4 shows the gradual typing rules for these three new constructs. In the (GAssign2) we allow the type of the right-hand side to differ from the type in the left-hand's reference, but require the types to be compatible. This is similar to the (GApp2) rule for function application.

We do not change the definition of the consistency relation, which means that references types are invariant with respect to consistency. The reflexive axiom $\tau \sim \tau$ implies that $\mathit{ref}\ \tau \sim \mathit{ref}\ \tau$. The situation is analogous to that of the combination of references with subtyping [32]: allowing variance under reference types compromises type safety. The following program demonstrates how a covariant rule for reference types would allow type errors to go uncaught by the type system.

```
let r1 = ref (λ y. y) in
let r2 : ref ? = r1 in
  r2 ← 1;
  !r1 2
```

The reference r1 is initialized with a function, and then r2 is aliased to r1, using the covariance to allow the change in type to ref ?. We can then write an integer into the cell pointed to by r2 (and by r1). The subsequent attempt to apply the contents of r1 as if it were a function fails at runtime.

5. Run-time semantics

We define the semantics for $\lambda^{?}_{\rightarrow}$ in two steps. We first define a cast insertion translation from $\lambda^{?}_{\rightarrow}$ to an intermediate language with explicit casts which we call $\lambda^{\langle\tau\rangle}_{\rightarrow}$. We then define a call-by-value operational semantics for $\lambda^{\langle\tau\rangle}_{\rightarrow}$. The explicit casts have the syntactic form $\langle\tau\rangle e$ where τ is the target type. When e evaluates to v, the cast will check that the type of v is consistent with τ and then produce a value based on v that has the type τ. If the type of v is inconsistent with τ, the cast produces a *CastError*. The intuition behind this kind of cast is that it reinterprets a value to have a different type either by adding or removing type information.

The syntax of $\lambda^{\langle\tau\rangle}_{\rightarrow}$ extends that of $\lambda^{?}_{\rightarrow}$ by adding a cast expression.

Syntax of the intermediate language. $e \in \lambda^{\langle\tau\rangle}_{\rightarrow}$

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \langle\tau\rangle e$$

5.1 Translation to $\lambda^{\langle\tau\rangle}_{\rightarrow}$.

The cast insertion judgment, defined in Figure 5, has the form $\Gamma \vdash e \Rightarrow e' : \tau$ and mimics the structure of our gradual typing judgment of Figure 2. It is trivial to show that these two judgments accept the same set of terms. We presented the gradual typing judgment separately to provide an uncluttered *specification* of well-typed terms. In Figure 5, the rules for variables, constants, and functions are straightforward. The first rule for application (CApp1) handles the case when the function has type ? and inserts a cast to $\tau_2 \rightarrow ?$ where τ_2 is the argument's type. The second rule for application (CApp2) handles the case when the function's type is known and the argument type differs from the parameter type, but is consistent. In this case the argument is cast to the parameter type τ. We could have instead cast the function; the choice was arbitrary. The third rule for application (CApp3) handles the case when the function type is known and the argument's type is identical to the parameter type. No casts are needed in this case. The rules for reference assignment are similar to the rules for application. However, for CAssign2 the choice to cast the argument and not the reference is because we need references to be invariant to preserve type soundness.

Next we define a type system for the intermediate language $\lambda^{\langle\tau\rangle}_{\rightarrow}$. The typing judgment has the form $\Gamma|\Sigma \vdash e : \tau$. The Σ is a store typing: it assigns types to memory locations. The type system, defined in Figure 6, extends the STLC with a rule for explicit casts. The rule (TCast) requires the expression e to have a type consistent with the target type τ.

The inversion lemmas for $\lambda^{\langle\tau\rangle}_{\rightarrow}$ are straightforward.

Lemma 1 (Inversion on typing rules.)**.**

1. If $\Gamma \mid \Sigma \vdash x : \tau$ then $\Gamma\, x = \lfloor\tau\rfloor$.
2. If $\Gamma \mid \Sigma \vdash c : \tau$ then $\Delta\, c = \tau$.
3. If $\Gamma \mid \Sigma \vdash \lambda\, x{:}\sigma.\ e : \tau$ then $\exists\, \tau'.\ \tau = \sigma \rightarrow \tau'$.
4. If $\Gamma \mid \Sigma \vdash e_1\ e_2 : \tau'$ then $\exists \tau.\ \Gamma \mid \Sigma \vdash e_1 : \tau \rightarrow \tau' \wedge \Gamma \mid \Sigma \vdash e_2 : \tau$.
5. If $\Gamma \mid \Sigma \vdash \langle\sigma\rangle\ e : \tau$ then $\exists\, \tau'.\ \Gamma \mid \Sigma \vdash e : \tau' \wedge \sigma = \tau \wedge \tau' \sim \sigma$.
6. If $\Gamma \mid \Sigma \vdash \mathit{ref}\ e : \mathit{ref}\ \tau$ then $\Gamma \mid \Sigma \vdash e : \tau$.
7. If $\Gamma \mid \Sigma \vdash !e : \tau$ then $\Gamma \mid \Sigma \vdash e : \mathit{ref}\ \tau$.
8. If $\Gamma \mid \Sigma \vdash e_1 \leftarrow e_2 : \mathit{ref}\ \tau$ then $\Gamma \mid \Sigma \vdash e_1 : \mathit{ref}\ \tau \wedge \Gamma \mid \Sigma \vdash e_2 : \tau$.

Figure 5. Cast Insertion

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

(CVar) $$\frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \vdash x \Rightarrow x : \tau}$$

(CConst) $$\frac{\Delta\, c = \tau}{\Gamma \vdash c \Rightarrow c : \tau}$$

(CLam) $$\frac{\Gamma(x \mapsto \sigma) \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \lambda\, x{:}\sigma.\, e \Rightarrow \lambda\, x{:}\sigma.\, e' : \sigma \to \tau}$$

(CApp1) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1\, e_2 \Rightarrow (\langle \tau_2 \to ? \rangle\, e'_1)\, e'_2 : ?}$$

(CApp2) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2 \qquad \tau_2 \neq \tau \qquad \tau_2 \sim \tau}{\Gamma \vdash e_1\, e_2 \Rightarrow e'_1\, (\langle \tau \rangle\, e'_2) : \tau'}$$

(CApp3) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1\, e_2 \Rightarrow e'_1\, e'_2 : \tau'}$$

(CRef) $$\frac{\Gamma \vdash e \Rightarrow e' : \tau}{\Gamma \vdash \mathit{ref}\, e \Rightarrow \mathit{ref}\, e' : \mathit{ref}\, \tau}$$

(CDeref1) $$\frac{\Gamma \vdash e \Rightarrow e' : ?}{\Gamma \vdash\, !e \Rightarrow\, !(\langle \mathit{ref}\, ? \rangle\, e') : ?}$$

(CDeref2) $$\frac{\Gamma \vdash e \Rightarrow e' : \mathit{ref}\, \tau}{\Gamma \vdash\, !e \Rightarrow\, !e' : \tau}$$

(CAssign1) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : ? \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau_2}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow (\langle \mathit{ref}\, \tau_2 \rangle\, e'_1) \leftarrow e'_2 : \mathit{ref}\, \tau_2}$$

(CAssign2) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \mathit{ref}\, \tau \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \sigma \qquad \sigma \neq \tau \qquad \sigma \sim \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow (\langle \tau \rangle\, e'_2) : \mathit{ref}\, \tau}$$

(CAssign3) $$\frac{\Gamma \vdash e_1 \Rightarrow e'_1 : \mathit{ref}\, \tau \qquad \Gamma \vdash e_2 \Rightarrow e'_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 \Rightarrow e'_1 \leftarrow e'_2 : \mathit{ref}\, \tau}$$

Proof Sketch. They are proved by case analysis on the type rules. □

The type system for $\lambda_{\to}^{\langle\tau\rangle}$ is deterministic: it assigns a unique type to an expression given a fixed environment.

Lemma 2 (Unique typing)**.** If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e : \tau'$ then $\tau = \tau'$.

Proof Sketch. The proof is by induction on the typing derivation and uses the inversion lemmas. □

The cast insertion translation, if successful, produces well-typed terms of $\lambda_{\to}^{\langle\tau\rangle}$.

Lemma 3. If $\Gamma \vdash e \Rightarrow e' : \tau$ then $\Gamma \mid \emptyset \vdash e' : \tau$.

Figure 6. Type system for the intermediate language $\lambda_{\to}^{\langle\tau\rangle}$

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

(TVar) $$\frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \mid \Sigma \vdash x : \tau}$$

(TConst) $$\frac{\Delta\, c = \tau}{\Gamma \mid \Sigma \vdash c : \tau}$$

(TLam) $$\frac{\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \lambda\, x{:}\sigma.\, e : \sigma \to \tau}$$

(TApp) $$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau \to \tau' \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1\, e_2 : \tau'}$$

(TCast) $$\frac{\Gamma \mid \Sigma \vdash e : \sigma \qquad \sigma \sim \tau}{\Gamma \mid \Sigma \vdash \langle \tau \rangle\, e : \tau}$$

(TRef) $$\frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \mathit{ref}\, e : \mathit{ref}\, \tau}$$

(TDeref) $$\frac{\Gamma \mid \Sigma \vdash e : \mathit{ref}\, \tau}{\Gamma \mid \Sigma \vdash\, !e : \tau}$$

(TAssign) $$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathit{ref}\, \tau \qquad \Gamma \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash e_1 \leftarrow e_2 : \mathit{ref}\, \tau}$$

(TLoc) $$\frac{\Sigma\, l = \lfloor \tau \rfloor}{\Gamma \mid \Sigma \vdash l : \mathit{ref}\, \tau}$$

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

When applied to terms of $\lambda_{\to}$, the translation is the identity function, i.e., no casts are inserted. [1]

Lemma 4. If $\emptyset \vdash e \Rightarrow e' : \tau$ and $e \in \lambda_{\to}$ then $e = e'$.

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

When applied to terms of the untyped λ-calculus, the translation inserts just those casts necessary to prevent type errors from occuring at run-time, such as applying a non-function.

5.2 Run-time semantics of $\lambda_{\to}^{\langle\tau\rangle}$.

The following grammar describes the results of evaluation: the result is either a value or an error, where values are either a simple value (variables, constants, functions, and locations) or a simple value enclosed in a single cast, which serves as a syntacical representation of boxed values.

[1] This lemma is for closed terms (this missing Γ means an empty environment). A similar lemma is true of open terms, but we do not need the lemma for open terms and the statement is more complicated because there are conditions on the environment.

Values, Errors, and Results

Locations	$l \in \mathbb{L}$		
Simple Values	$s \in \mathbb{S}$	$::=$	$x \mid c \mid \lambda x : \tau.e \mid l$
Values	$v \in \mathbb{V}$	$::=$	$s \mid \langle ? \rangle s$
Errors	ε	$::=$	*CastError* $\mid$ *TypeError* $\mid$ *KillError*
Results	r	$::=$	$v \mid \varepsilon$

It is useful to distinguish two different kinds of run-time type errors. In weakly typed languages, type errors result in undefined behavior, such as causing a segmentation fault or allowing a hacker to create a buffer overflow. We model this kind of type error with *TypeError*. In strongly-typed dynamic languages, there may still be type errors, but they are caught by the run-time system and do not cause undefined behavior. They typically cause the program to terminate or else raise an exception. We model this kind of type error with *CastError*. The *KillError* is a technicality pertaining to the type safety proof that allows us to prove a form of "progress" in the setting of a big-step semantics.

We define simple function values (*SimpleFunVal*) to contain lambda abstractions and functional constants (such as succ), and function values (*FunVal*) include simple function values and simple function values cast to ?.

As mentioned in Section 1, the Canonical Forms Lemma is of particular interest due to its implications for performance. When an expression has either ground or function type (not ?) the kind of resulting value is fixed, and a compiler may use an efficient unboxed representation. For example, if an expression has type int, then it will evaluate to a value of type int (by the forthcoming Soundness Lemma 8) and then the Canonical Forms Lemma tells us that the value must be an integer.

Lemma 5 (Canonical Forms)**.**

- If $\emptyset \mid \Sigma \vdash v : \mathit{int}$ and $v \in \mathbb{V}$ then $\exists n.\ v = n$.
- If $\emptyset \mid \Sigma \vdash v : \mathit{bool}$ and $v \in \mathbb{V}$ then $\exists b.\ v = b$.
- If $\emptyset \mid \Sigma \vdash v : ?$ and $v \in \mathbb{V}$ then $\exists v'.\ v = \langle ? \rangle\ v' \wedge v' \in \mathbb{S}$.
- If $\emptyset \mid \Sigma \vdash v : \tau \to \tau'$ and $v \in \mathbb{V}$ then $v \in \mathit{SimpleFunVal}$.
- If $\emptyset \mid \Sigma \vdash v : \mathit{ref}\ \tau$ and $v \in \mathbb{V}$ then $\exists l.\ v = l \wedge \Sigma\ l = \lfloor \tau \rfloor$.

Proof Sketch. They are proved using the inversion lemmas and case analysis on values. □

We define the run-time semantics for $\lambda^{\langle \tau \rangle}_{\rightarrow}$ in big-step style with substitution and not environments. Substitution, written $[x := e]e$, is formalized in the style of Curry [3], where bound variables are α-renamed during substitution to avoid the capture of free variables.

The evaluation judgment has the form $e \hookrightarrow_n r$, where e evaluates to the result r with a derivation depth of n. The derivation depth is used to force termination so that derivations can be constructed for otherwise non-terminating programs [11]. The n-depth evaluation allows Lemma 8 (Soundness) to distinguish between terminating and non-terminating programs. We will say more about this when we get to Lemma 8.

The evaluation rules, shown in Figures 7 and 8, are the standard call-by-value rules for the λ-calculus [33] with additional rules for casts and a special termination rule. We parameterize the semantics over the function δ which defines the behavior of functional constants and is used in rule (EDELTA). The helper function *unbox* removes an enclosing cast from a value, if there is one.

$$\begin{aligned} \mathit{unbox}\ s &= s \\ \mathit{unbox}\ (\langle \tau \rangle\ s) &= s \end{aligned}$$

The evaluation rules treat the cast expression like a boxed, or tagged, value. It is straightforward to define a lower-level semantics that explicitly tags every value with its type (the full type, not just the top level constructor) and then uses these type representations instead of the typing judgment $\emptyset \mid \emptyset \vdash \mathit{unbox}\ v : \tau$, as in the rule (ECSTG).

There is a separate cast rule for each kind of target type. The rule (ECSTG) handles the case of casting to a ground type. The cast is removed provided the run-time type exactly matches the target type. The rule (ECSTF) handles the case of casting to a function type. If the run-time type is consistent with the target type, the cast is removed and the inner value is wrapped inside a new function that inserts casts to produce a well-typed value of the appropriate type. This rule is inspired by the work on semantic casts [13, 14, 15], though the rule may look slightly different because the casts used in this paper are annotated with the target type only and not also with the source type. The rule (ECSTR) handles the case of casting to a reference type. The run-time type must exactly match the target type. The rule (ECSTU) handles the case of casting to ? and ensures that nested casts are collapsed to a single cast. The rule (ECSTE) handles the case when the run-time type is not consistent with the target type and produces a *CastError*.

Because the target types of casts are static, the cast form could be replaced by a cast for each type, acting as injection to ? and projection to ground and function types. However, this would complicate the rules, especially the rule for casting to a function type.

The rule (EKILL) terminates evaluation when the derivation depth counter reaches zero.

5.3 Examples

Consider once again the following program and assume the succ constant has the type **number** → **number**.

```
((λ (x) (succ x)) #t)
```

The cast insertion judgement transforms this term into the following term.

```
((λ (x : ?) (succ ⟨number⟩x)) ⟨?⟩#t)
```

Evaluation then proceeds, applying the function to its argument, substituting ⟨?⟩#t for x.

```
(succ ⟨number⟩⟨?⟩#t)
```

The type of #t is **boolean**, which is not consistent with **number**, so the rule (ECSTE) applies and the result is a cast error.

```
CastError
```

Next, we look at an example that uses first-class functions.

```
((λ (f : ? → number) (f 1))
 (λ (x : number) (succ x)))
```

Cast insertion results in the following program.

```
((λ (f : ? → number) (f ⟨?⟩1))
 ⟨? → number⟩(λ (x : number) (succ x)))
```

We apply the cast to the function, creating a wrapper function.

```
((λ (f : ? → number) (f ⟨?⟩1))
 (λ (z : ?) ⟨number⟩((λ (x : number) (succ x)) ⟨number⟩z)))
```

Function application results in the following

Figure 7. Evaluation

$$\boxed{e|\mu \hookrightarrow_n r|\mu}$$

Casting

$$\text{(ECstG)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \gamma}{\langle\gamma\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{unbox}\ v \mid \mu'}$$

$$\text{(ECstF)}\quad \frac{\begin{array}{c} e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \tau \to \tau' \\ \tau \to \tau' \sim \sigma \to \sigma' \qquad z = \mathit{max} v\ v + 1 \end{array}}{\langle\sigma \to \sigma'\rangle\ e \mid \mu \hookrightarrow_{n+1} \lambda\, z{:}\sigma.\ (\langle\sigma'\rangle\ (\mathit{unbox}\ v\ (\langle\tau\rangle\ z))) \mid \mu'}$$

$$\text{(ECstR)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \mathit{ref}\ \tau}{\langle \mathit{ref}\ \tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{unbox}\ v \mid \mu'}$$

$$\text{(ECstU)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu'}{\langle ?\rangle\ e \mid \mu \hookrightarrow_{n+1} \langle ?\rangle\ \mathit{unbox}\ v \mid \mu'}$$

Functions and constants

$$\text{(ELam)}\quad \frac{0 < n}{\lambda\, x{:}\tau.\ e \mid \mu \hookrightarrow_n \lambda\, x{:}\tau.\ e \mid \mu}$$

$$\text{(EApp)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n \lambda\, x{:}\tau.\ e_3 \mid \mu_2 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x{:=}v_2]e_3 \mid \mu_3 \hookrightarrow_n v_3 \mid \mu_4 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} v_3 \mid \mu_4}$$

$$\text{(EConst)}\quad \frac{0 < n}{c \mid \mu \hookrightarrow_n c \mid \mu}$$

$$\text{(EDelta)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n c_1 \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n c_2 \mid \mu_3}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \delta\ c_1\ c_2 \mid \mu_3}$$

References

$$\text{(ERef)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad l \notin \mathit{dom}\ \mu'}{\mathit{ref}\ e \mid \mu \hookrightarrow_{n+1} l \mid \mu'(l \mapsto v)}$$

$$\text{(EDeref)}\quad \frac{e \mid \mu \hookrightarrow_n l \mid \mu' \qquad \mu'\, l = \lfloor v \rfloor}{!e \mid \mu \hookrightarrow_{n+1} v \mid \mu'}$$

$$\text{(EAssign)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n l \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n v \mid \mu_3}{e_1 \leftarrow e_2 \mid \mu_1 \hookrightarrow_{n+1} l \mid \mu_3(l \mapsto v)}$$

$$\text{(ELoc)}\quad \frac{0 < n}{l \mid \mu \hookrightarrow_n l \mid \mu}$$

Figure 8. Evaluation (Errors)

$$\text{(ECstE)}\quad \frac{\begin{array}{c} e \mid \mu \hookrightarrow_n v \mid \mu' \\ \emptyset \mid \Sigma \vdash \mathit{unbox}\ v : \sigma \qquad (\sigma, \tau) \notin \mathit{op} \sim \end{array}}{\langle\tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \mathit{CastError} \mid \mu'}$$

$$\text{(EKill)}\quad e \mid \mu \hookrightarrow_0 \mathit{KillError} \mid \mu$$

$$\text{(EVarT)}\quad \frac{0 < n}{x \mid \mu \hookrightarrow_n \mathit{TypeError} \mid \mu}$$

$$\text{(EAppT)}\quad \frac{e_1 \mid \mu \hookrightarrow_n v_1 \mid \mu' \qquad v_1 \notin \mathit{FunVal}}{e_1\ e_2 \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

$$\text{(ECstP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{\langle\tau\rangle\ e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAppP1)}\quad \frac{e_1 \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{e_1\ e_2 \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAppP2)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n v_1 \mid \mu_2 \\ v_1 \in \mathit{FunVal} \qquad e_2 \mid \mu_2 \hookrightarrow_n \varepsilon \mid \mu_3 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_3}$$

$$\text{(EAppP3)}\quad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n \lambda\, x{:}\tau.\ e_3 \mid \mu_2 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x{:=}v_2]e_3 \mid \mu_3 \hookrightarrow_n \varepsilon \mid \mu_4 \end{array}}{e_1\ e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_4}$$

$$\text{(ERefP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{\mathit{ref}\ e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EDerefP)}\quad \frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{!e \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAssignP1)}\quad \frac{e_1 \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{e_1 \leftarrow e_2 \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\text{(EAssignP2)}\quad \frac{e_1 \mid \mu_1 \hookrightarrow_n l \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n \varepsilon \mid \mu_3}{e_1 \leftarrow e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_3}$$

$$\text{(EDerefT)}\quad \frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \nexists\, l.\ v = l}{!e \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

$$\text{(EAssignT)}\quad \frac{e_1 \mid \mu \hookrightarrow_n v \mid \mu' \qquad \nexists\, l.\ v = l}{e_1 \leftarrow e_2 \mid \mu \hookrightarrow_{n+1} \mathit{TypeError} \mid \mu'}$$

$((\lambda\ (z : ?)\ \langle \textbf{number} \rangle((\lambda\ (x : \textbf{number})\ (\text{succ}\ x))\ \langle \textbf{number} \rangle z))\ \langle ? \rangle 1)$

and then another function application gives us

$\langle \textbf{number} \rangle((\lambda\ (x : \textbf{number})\ (\text{succ}\ x))\ \langle \textbf{number} \rangle \langle ? \rangle 1)$

We then apply the cast rule for ground types (ECSTG).

$\langle \textbf{number} \rangle((\lambda\ (x : \textbf{number})\ (\text{succ}\ x))\ 1)$

followed by another function application:

$\langle \textbf{number} \rangle(\text{succ}\ 1)$

Then by (EDELTA) we have

$\langle \textbf{number} \rangle 2$

and by (ECSTG) we finally have the result

2

5.4 Type Safety

Towards proving type safety we prove the usual lemmas. First, environment expansion and contraction does not change typing derivations. Also, changing the store typing environment does not change the typing derivations as long as the new store typing agrees with the old one. The function *Vars* returns the free and bound variables of an expression.

Lemma 6 (Environment Expansion and Contraction)**.**

- If $\Gamma \mid \Sigma \vdash e : \tau$ and $x \notin \mathit{Vars}\ e$ then $\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau$.
- If $\Gamma(y \mapsto \nu) \mid \Sigma \vdash e : \tau$ and $y \notin \mathit{Vars}\ e$ then $\Gamma \mid \Sigma \vdash e : \tau$.
- If $\Gamma \mid \Sigma \vdash e : \tau$ and $\bigwedge l.$ If $l \in \mathit{dom}\ \Sigma$ then $\Sigma' l = \Sigma\ l.$ then $\Gamma \mid \Sigma' \vdash e : \tau$.

Proof Sketch. These properties are proved by induction on the typing derivation. □

Also, substitution does not change the type of an expression.

Lemma 7 (Substitution preserves typing)**.** If $\Gamma(x \mapsto \sigma) \mid \Sigma \vdash e : \tau$ and $\Gamma \mid \Sigma \vdash e' : \sigma$ then $\Gamma \mid \Sigma \vdash [x{:=}e']e : \tau$.

Proof Sketch. The proof is by strong induction on the size of the expression e, using the inversion and environment expansion lemmas. □

Definition 1. *The store typing judgment, written $\Gamma|\Sigma \models \mu$, holds when the domains of Σ and μ are equal and when for every location l in the domain of Σ there exists a type τ such that $\Gamma|\Sigma \vdash \mu(l) : \tau$.*

Next we prove that n-depth evaluation for the intermediate language $\lambda_{\rightarrow}^{\langle\tau\rangle}$ is sound. Informally, this lemma says that evaluation produces either a value of the appropriate type, a cast error, or *KillError* (because evaluation is cut short), but never a type error. The placement of $e \mid \mu \hookrightarrow_n r \mid \mu'$ in the conclusion of the lemma proves that our evaluation rules are complete, analogous to a progress lemma for small-step semantics. This placement would normally be a naive mistake because not all programs terminate. However, by using n-depth evaluation, we can construct a judgment regardless of whether the program is non-terminating because evaluation is always cut short if the derivation depth exceeds n. But does this lemma handle all terminating programs? The lemma is (implicitly) universally quantified over the evaluation depth n. For every program that terminates there is a depth that will allow it to terminate, and this lemma will hold for that depth. Thus, this lemma applies to all terminating programs and does not apply to non-terminating program, as we intend. We learned of this technique from Ernst, Ostermann, and Cook [11], but its origins go back at least to Volpano and Smith [41].

Lemma 8 (Soundness of evaluation)**.** If $\emptyset \mid \Sigma \vdash e : \tau \wedge \emptyset \mid \Sigma \models \mu$ then $\exists\, r\, \mu'\, \Sigma'.\ e \mid \mu \hookrightarrow_n r \mid \mu' \wedge \emptyset \mid \Sigma' \models \mu' \wedge (\forall l.\ l \in \mathit{dom}\ \Sigma \longrightarrow \Sigma'\ l = \Sigma\ l) \wedge ((\exists v.\ r = v \wedge v \in \mathbb{V} \wedge \emptyset \mid \Sigma' \vdash v : \tau) \vee r = \mathit{CastError} \vee r = \mathit{KillError})$.

Proof. The proof is by strong induction on the evaluation depth. We then perform case analysis on the final step of the typing judgment. The case for function application uses the substitution lemma and the case for casts uses environment expansion. The cases for references and assign use the lemma for changing the store typing. The inversion lemmas are used throughout. □

Theorem 2 (Type safety)**.** If $\emptyset \vdash e \Rightarrow e' : \tau$ then $\exists\, r\, \mu\, \Sigma.\ e' \mid \emptyset \hookrightarrow_n r \mid \mu \wedge ((\exists v.\ r = v \wedge v \in \mathbb{V} \wedge \emptyset \mid \Sigma \vdash v : \tau) \vee r = \mathit{CastError} \vee r = \mathit{KillError})$.

Proof. Apply Lemma 3 and then Lemma 8. □

6. Relation to Dynamic of Abadi et al.

We defined the semantics for $\lambda_{\rightarrow}^{?}$ with a translation to $\lambda_{\rightarrow}^{\langle\tau\rangle}$, a language with explicit casts. Perhaps a more obvious choice for intermediate language would be the pre-existing language of explicit casts of Abadi et. all [1]. However, there does not seem to be a straightforward translation from $\lambda_{\rightarrow}^{\langle\tau\rangle}$ to their language. Consider the evaluation rule (ECSTF) and how that functionality might be implemented in terms of typecase. The parameter z must be cast to τ, which is not known statically but only dynamically. To implement this cast we would need to dispatch based on τ, perhaps with a typecase. However, typecase must be applied to a value, and there is no way for us to obtain a value of type τ from a value of type $\tau \rightarrow \tau'$. Quoting from [1]:

> Neither tostring nor typetostring quite does its job: for example, when tostring gets to a function, it stops without giving any more information about the function. It can do no better, given the mechanisms we have described, since there is no effective way to get from a function value to an element of its domain or codomain.

Of course, if their language were to be extended with a construct for performing case analysis on types, such as the typerec of Harper and Morrisett [19], it would be straightforward to implement the appropriate casting behavior.

7. Related Work

Several programming languages provide gradual typing to some degree, such as Cecil [8], Boo [10], extensions to Visual Basic.NET and C# proposed by Meijer and Drayton [26], extensions to Java proposed by Gray et al. [17], and the Bigloo [6, 36] dialect of Scheme [24]. This paper formalizes a type system that provides a theoretical foundation for these languages.

Common LISP [23] and Dylan [12, 37] include optional type annotations, but the annotations are not used for type checking, they are used to improve performance.

Cartwright and Fagan's Soft Typing [7] improves the performance of dynamically typed languages by inferring types and removing the associated run-time dispatching. They do not focus on statically

catching type errors, as we do here, and do not study a source language with optional type annotations.

Anderson and Drossopoulou formalize BabyJ [2], an object-oriented language inspired by JavaScript. BabyJ has a nominal type system, so types are class names and the permissive type $*$. In the type rules for BabyJ, whenever equality on types would normally be used, they instead use the relation $\tau_1 \approx \tau_2$ which holds whenever τ_1 and τ_2 are the same name, or when at least one of them is the permissive type $*$. Our unknown type ? is similar to the permissive type $*$, however, the setting of our work is a structural type system and our type compatibility relation $\sim$ takes into account function types.

Riely and Hennessy [35] 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.

Gradual typing is syntactically similar to type inferencing [9, 21, 27]: both approaches allow type annotations to be omitted. However, with type inference, the type system tries to reconstruct what the type annotations should be, and if it cannot, rejects the program. In contrast, a gradual type system accepts that it does not know certain types and inserts run-time casts.

Henglein [20] presents a translation from untyped λ-terms to a coercion calculus with explicit casts. These casts make explicit the tagging, untagging, and tag-checking operations that occur during the execution of a language with latent (dynamic) typing. Henglein's coercion calculus seems to be closely related to our $\lambda^{\langle\tau\rangle}_{\rightarrow}$ but we have not yet formalized the relation. Henglein does not study a source language with partially typed terms with a static type system, as we do here. Instead, his source language is a dynamically typed language.

Bracha [4] defines *optional type systems* as type systems that do not affect the semantics of the language and where type annotations are optional. Bracha cites Strongtalk [5] as an example of an optional type system, however, that work does not define a formal type system or describe how omitted type annotations are treated.

Ou et. all. [31] 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 analogous to the combination of dynamic typing and static typing presented in this paper.

Flanagan [15] introduces Hybrid Type Checking, which 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 also 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 the consistency relation.

Gronski, Knowles, Tomb, Freund, and Flanagan [18] developed the Sage language which provides Hybrid Type Checking and also a Dynamic type with implicit (run-time checked) down-casts. Surprisingly, the Sage type system does not allow implicit down-casts from Dynamic, whereas the Sage type checking (and compilation) algorithm does allow implicit down-casts. It may be that the given type system was intended to characterize the output of compilation (though it is missing a rule for cast), but then a type system for the source language remains to be defined. The Sage technical report [18] does not include a result such as Theorem 1 of this paper to show that the type system catches all type errors for fully annotated programs, which is a tricky property to achieve in the presence of a Dynamic type with implicit down-casts.

There are many interesting issues regarding efficient representations for values in a language that mixes static and dynamic typing. The issues are the same as for parametric polymorphism (dynamic typing is just a different kind of polymorphism). Leroy [25] discusses the use of mixing boxed and unboxed representations and such an approach is also possible for our gradual type system. Shao [38] further improves on Leroy's mixed approach by showing how it can be combined with the type-passing approach of Harper and Morrisett [19] and thereby provide support for recursive and mutable types.

8. Conclusion

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 than static 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 development of a program in a dynamic language and then translate to a static language midway through development. 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 while staying within the same language.

In this paper we present the formal definition of the language $\lambda^{?}_{\rightarrow}$, including its static and dynamic semantics. This language captures the key ingredients for implementing gradual typing in functional languages. The language $\lambda^{?}_{\rightarrow}$ provides the flexibility of dynamically typed languages when type annotations are omitted by the programmer and provides the benefits of static checking when all function parameters are annotated, including the safety guarantees (Theorem 1) and the time and space efficiency (Lemma 5). Furthermore, the cost of dynamism is "pay-as-you-go", so partially annotated programs enjoy the benefits of static typing to the degree that they are annotated. We prove type safety for $\lambda^{?}_{\rightarrow}$ (Theorem 2); the type system prevents type violations from occurring at run-time, either by catching the errors statically or by catching them dynamically with a cast exception. The type system and run-time semantics of $\lambda^{?}_{\rightarrow}$ is relatively straightforward, so it is suitable for practical languages.

As future work, we intend to investigate the interaction between our gradual type system and types such as lists, arrays, algebraic data types, and implicit coercions between types, such as the types in Scheme's numerical tower. We also plan to investigate the interaction between gradual typing and parametric polymorphism [16, 34] and Hindley-Milner inference [9, 21, 27]. We have implemented and tested an interpreter for the $\lambda^{?}_{\rightarrow}$ calculus. As future work we intend to incorporate gradual typing as presented here 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 Emir Pasalic the members of the Resource Aware Programming Laboratory for reading drafts and suggesting improvements. 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.

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. Elsevier, 2003.

[3] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[12] 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.

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

[14] 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.

[15] 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.

[16] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[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. Technical report, University of California, Santa Cruz, 2006.

[19] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.

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

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

[22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[23] 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.

[24] 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.

[25] X. Leroy. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM Press.

[26] 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.

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

[28] T. Nipkow. Structured proofs in Isar/HOL. In *TYPES*, number 2646 in LNCS, 2002.

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

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

[31] 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.

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

[33] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[34] 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).

[35] 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.

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

[37] 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.

[38] Z. Shao. Flexible representation analysis. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 85–98, New York, NY, USA, 1997. ACM Press.

[39] J. Siek and W. Taha. Gradual typing: Isabelle/isar formalization. Technical Report TR06-874, Rice University, Houston, Texas, 2006.

[40] 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.

[41] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW'97: 10th Computer Security Foundations Workshop*, volume 00, page 156, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

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