GRADUAL TYPING FOR FUNCTIONAL LANGUAGES

Jaseem Abid

@jaseemabid

WHAT ARE WE TALKING ABOUT ?

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.

EH?

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.

GRADUAL TYPING

PRELUDE

- Type Systems
- Lambda Calculus, Untyped & Simply Typed
- Scheme
- PL Notation

WHAT WILL BE COVERED TODAY?

- Gradual typing
- Current state of the implementations

Static and dynamic type systems have well known strengths and weaknesses, and each is better suited for different programming tasks.

We can do better.

PROBLEM STATEMENT

	Weak	Strong
Static	Go, Java	Haskell
Dynamic	JavaScript	Scheme, Python

DYNAMIC AND WEAK

```
> 42 + "life"
'42life'
```

DYNAMIC BUT STRONG

```
Python 2.7.11 (default, Jan 12 2016, 12:18:17)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> 42 + "life"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of

LINES ARE BLURRING

THERE IS ABSOLUTELY NO ADVANTAGE IN USING A DYNAMICALLY TYPED LANGUAGE IF VERBOSITY AND FRICTION IS REMOVED FROM STATICALLY TYPED LANGUAGES

My own rants

Intended for submission to the *Revival of Dynamic Languages*

Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages

Erik Meijer and Peter Drayton
Microsoft Corporation

Abstract

This paper argues that we should seek the golden middle way between dynamically and statically typed languages.

1 Introduction

<NOTE to="reader"> *Please note that this paper is still very much work in progress and as such the presentation is unpolished and possibly incoherent. Obviously many citations to related and relevant work are missing. We did however do our best to make it provocative.* </NOTE>

Advocates of static typing argue that the advantages of static typing include earlier detection of programming mistakes (e.g. preventing adding an integer to a boolean), better documentation in the form of type signatures (e.g. incorporating number and types of arguments when resolving names), more opportunities for compiler optimizations (e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically), increased runtime efficiency (e.g. not all values need to carry a dynamic type), and a better design time developer experience (e.g. knowing the type of the receiver, the IDE can present a drop-down menu of all applicable members).

Advocates of dynamically typed languages argue that static typing is too rigid, and that the softness of dynamically languages makes them ideally suited for prototyping systems with changing or unknown requirements, or that interact with other systems that change unpredictably (data and application integration). Of course, dynamically typed languages are indispensable for dealing with truly dynamic program behavior such as method interception, dynamic loading, mobile code, runtime reflection, etc.

In the mother of all papers on scripting [16], John Ousterhout argues that statically typed systems programming languages make code less reusable, more verbose, not more safe, and less expressive than dynamically typed scripting languages. This argument is parroted literally by many proponents of dynamically typed scripting languages. We argue that this is a fallacy and falls into the same category as arguing that the essence of declarative programming is eliminating assignment. Or as John Hughes says [8], it is a logical impossibility to make a language more powerful by omitting features. Defending the fact that delaying all type-checking to runtime is a good thing, is playing ostrich tactics with the fact that errors should be caught as early in the development process as possible.

We are interesting in building data-intensive three-tiered enterprise applications [14]. Perhaps surprisingly, dynamism is

OPTIONAL VS GRADUAL

- Optional type annotations that are used to improve runtime performance but not to increase the amount of static checking.
- In a gradual type system, the type system is complete.
- If provided all type annotations, the program cannot fail with static type errors.

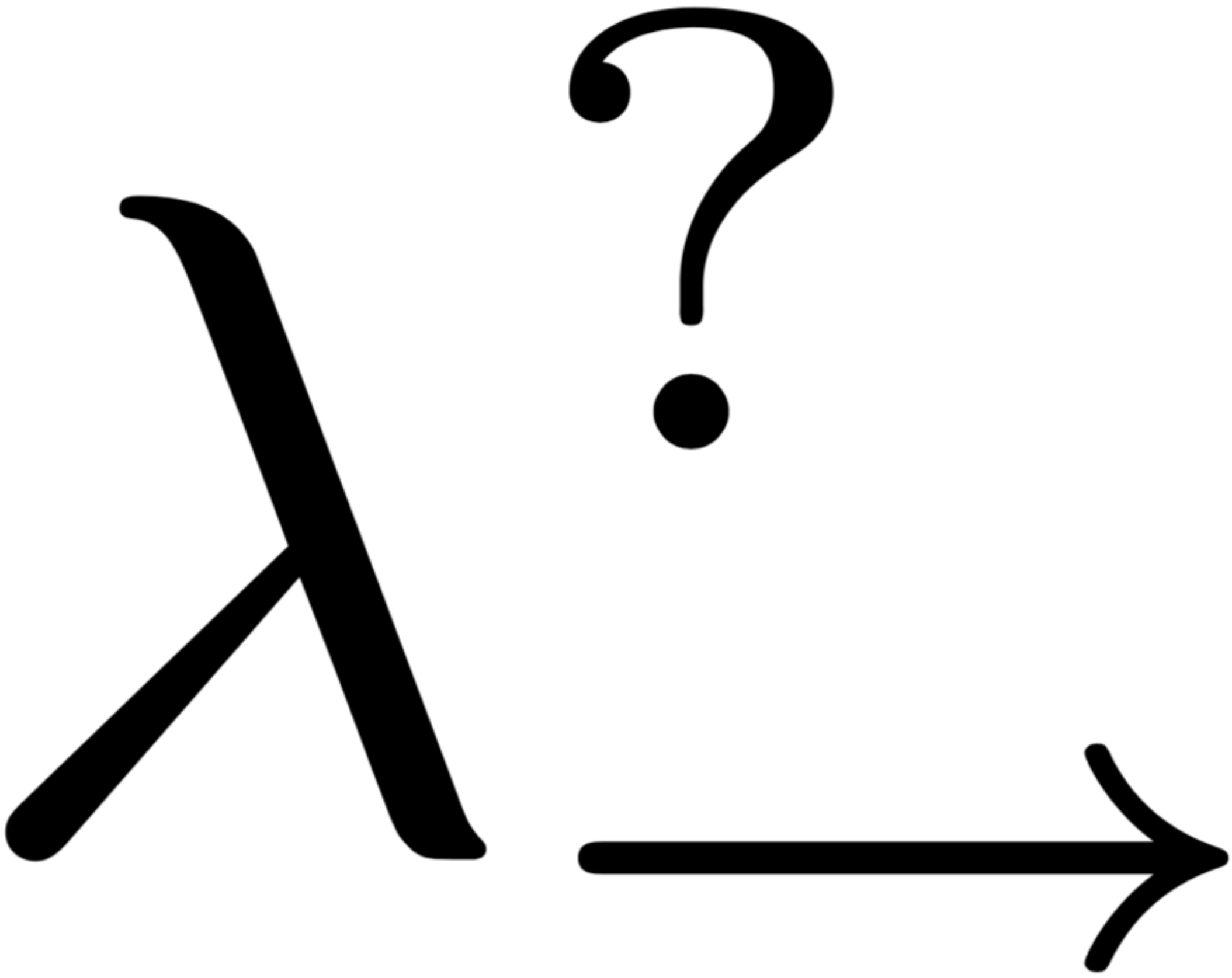
```
(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(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.

CONTRIBUTIONS

- A formal type system that supports gradual typing for functional languages.
- The flexibility of dynamically typed languages when type annotations are omitted & benefits of static checking when function parameters are annotated.



λ?

λ

- When applied to fully annotated terms, λ? is equivalent to λ→
- This property ensures that for fully annotated programs all type errors are caught at compile time.

THEOREM 1

We define the runtime semantics of λ? via a translation to λ→ with explicit casts.

RUN TIME SEMANTICS

λ<τ>

When applied to fully annotated terms, the translation does not insert casts, so the semantics exactly matches that of the simply typed calculus.

LEMMA 4

The translation preserves typing.

LEMMA 3

λ<τ> is type safe.

LEMMA 8

..and therefore λ? 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

The gradually typed λ calculus, λ?, is the λ→ extended with a type ? to represent dynamic types.

λ?

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	$::=$	$c \mid x \mid \lambda x{:}\tau.\ e \mid e\ e$
			$\lambda x.\ e \equiv \lambda x{:}?.\ e$

SYNTAX OF $e \in \lambda$?

- A procedure without a parameter type annotation is syntactic sugar for a procedure with parameter type ?
- All unknown parts are marked with ?
- The job of the static type system is to reject programs that have inconsistencies in the known parts of types.

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

```
((λ (x) (succ x)) #t)
```

map : (number → number) * number list → number list

```
(map (λ (x) (succ x)) (list 1 2 3))
```

Type Consistency $\tau \sim \tau$

(CREFL) $\tau \sim \tau$ (CFUN) $\dfrac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2}$

(CUNR) $\tau \sim ?$ (CUNL) $? \sim \tau$

REFLEXIVE, SYMMETRIC BUT NOT TRANSITIVE

Figure 2. A Gradual Type System

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

(GVAR)
$$\frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \vdash_G x : \tau}$$

(GCONST)
$$\frac{\Delta\, c = \tau}{\Gamma \vdash_G c : \tau}$$

(GLAM)
$$\frac{\Gamma(x \mapsto \sigma) \vdash_G e : \tau}{\Gamma \vdash_G \lambda\, x{:}\sigma.\, e : \sigma \to \tau}$$

(GAPP1)
$$\frac{\Gamma \vdash_G e_1 : ? \qquad \Gamma \vdash_G e_2 : \tau_2}{\Gamma \vdash_G e_1\ e_2 : ?}$$

(GAPP2)
$$\frac{\Gamma \vdash_G e_1 : \tau \to \tau' \qquad \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau}{\Gamma \vdash_G e_1\ e_2 : \tau'}$$

```
Δ          → Type env with constants
Γ          → Type env, λ :: e -> τ | ⊥
Γ\x        → Type env sans x
Γ,x:Γ      → Type env with x
Γ(x → σ) → Γ,x:σ
Inductive notation
```

http://siek.blogspot.in/2012/07/crash-course-on-notation-in-programming.html

---

CRASH COURSE ON NOTATION IN PROGRAMMING LANGUAGE THEORY

(succ “hi”)

RELATION TO THE UNTYPED λ CALCULUS

Equivalent for fully annotated terms

RELATION TO THE SIMPLY TYPED λ CALCULUS

Simply typed λ calculus minus ?

PROOF SKETCH

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

QUASI-STATIC TYPING

- Standard sub typing relation <:
- Dynamic type Ω
- Transitive types → Plausibility checking
- Upcast σ to Ω followed by downcast to τ
- Symmetric consistency vs anti symmetric sub typing

WELCOME TO THE REAL WORLD

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

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

- Define a cast insertion translation from λ? to an intermediate language with explicit casts, λ<τ>
- Call by value operational semantics for λ<τ>

RUNTIME SEMANTICS

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

Expressions	e	$::=$	$\ldots \mid \langle\tau\rangle e$

$$\Gamma \vdash e \Rightarrow e' : \tau$$

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

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

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

Type system for IR λ<τ>

Figure 6. Type system for the intermediate language $\lambda^{\langle\tau\rangle}_{\rightarrow}$

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

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

When applied to terms of $\lambda\!\!\rightarrow$, the translation is the identity function, i.e, no casts are inserted.

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.

Run time semantics of λ<τ>

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	$::=$	$v \mid \varepsilon$

```
((λ (x) (succ x)) #t)
((λ (x : ?) (succ <number>x)) <?>#t)
(succ <number><?>#t)
```

```
(ECSTE) => CastError
```

EVALUATION RULES

TYPE SAFETY

RELATED WORK

CONCLUSION

The Ins and Outs of Gradual Type Inference

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

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

Abstract

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

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

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

admit performance optimizations that the dynamically typed fragments do not. Gradual typing envisions a style of programming where dynamically typed programs can be *evolved* into statically typed programs by gradually trading off liberties in code structure for assurances of safety and performance.

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

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

Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen
Northeastern University, Boston, MA

Abstract

Programmers have come to embrace dynamically-typed languages for prototyping and delivering large and complex systems. When it comes to maintaining and evolving these systems, the lack of explicit static typing becomes a bottleneck. In response, researchers have explored the idea of gradually-typed programming languages which allow the incremental addition of type annotations to software written in one of these untyped languages. Some of these new, hybrid languages insert run-time checks at the boundary between typed and untyped code to establish type soundness for the overall system. With sound gradual typing, programmers can rely on the language implementation to provide meaningful error messages when type invariants are violated. While most research on sound gradual typing remains theoretical, the few emerging implementations suffer from performance overheads due to these checks. None of the publications on this topic comes with a comprehensive performance evaluation. Worse, a few report disastrous numbers.

In response, this paper proposes a method for evaluating the performance of gradually-typed programming languages. The method hinges on exploring the space of partial conversions from untyped to typed. For each benchmark, the performance of the different ver-

many cases, the systems start as innocent prototypes. Soon enough, though, they grow into complex, multi-module programs, at which point the engineers realize that they are facing a maintenance nightmare, mostly due to the lack of reliable type information.

Gradual typing [21, 26] proposes a language-based solution to this pressing software engineering problem. The idea is to extend the language so that programmers can incrementally equip programs with types. In contrast to optional typing, gradual typing provide programmers with soundness guarantees.

Realizing type soundness in this world requires run-time checks that watch out for potential impedance mismatches between the typed and untyped portions of the programs. The granularity of these checks determine the peformance overhead of gradual typing. To reduce the frequency of checks, *macro-level* gradual typing forces programmers to annotate entire modules with types and relies on behavioral contracts [12] between typed and untyped modules to enforce soundness. In contrast, *micro-level* gradual typing instead assigns an implicit type `Dyn` [1] to all unannotated parts of a program; type annotations can then be added to any declaration. The implementation must insert casts at the appropriate points in the code. Different language designs use slightly different

That's all Folks!