Gradual Typing

An Introduction and an Implementation in MIT/GNU Scheme

Ramana Nagasamudram, Khayyam Saleem

CS810 – Type Systems for Programming Languages

Outline

Topic

Gradual Typing

$\lambda^{?}_{\rightarrow}$

$\lambda^{\langle\tau\rangle}_{\rightarrow}$

Implementation

Concept

- Type system developed by Jeremy Siek and Walid Taha in 2006
- Allows some parts of a program to be dynamically typed and other parts to be statically typed
  - Determined by presence of type annotation added by programmer

Static Typing

- Process of verifying the type safety of a program based on analysis of a program's source code
- If a program passes a static type checker, then the program is guaranteed to satisfy some set of type safety properties for all possible inputs
- Type checking completed during compilation process

- Pros
  - catches bugs early
  - faster execution
  - improves modularity

- Cons
  - makes code more verbose
  - prevents program execution

Dynamic Typing

- Process of type-checking at run-time
- Associates each runtime object with a *type tag*

- Pros
  - offers flexibility
  - doesn't get "in the way" of execution
  - allows for typing based on runtime information

- Cons
  - cannot conclusively declare safety
  - errors may lie deep in subroutine calls
  - slower execution

Utility of Gradual Typing

- Gradual typing allows for type checks at compile-time for type errors in some parts of a program, directed by type annotations.
- Since it is tough to declare that static typing is universally better or worse than dynamic typing, gradual typing offers the programmer a choice, without requiring a change in language
- Provides a type system that:
  - allows programmers to choose the degree to which they want to annotate a program
  - allows programmers to use type annotations for static type checking as well as improving run-time performance
  - accepts programs written in a dynamically typed style
  - on completely annotated programs, behaves just like a static type system

Attempt with Subtyping

- Prior attempts at integrating static and dynamic typing tried to make the dynamic type be both the top and bottom of the subtype hierarchy.
- However, because subtyping is transitive, that results in every type becoming related to every other type, and so subtyping would no longer rule out any static type errors
- The addition of a second phase of plausibility checking to the type system did not completely solve this problem

Type Consistency

(CREFL) $\tau \sim \tau$

(CUNR) $\tau \sim ?$

(CFUN) $\dfrac{\sigma_1 \sim \tau_1 \quad \sigma_2 \sim \tau_2}{\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2}$

(CUNL) $? \sim \tau$

Topic

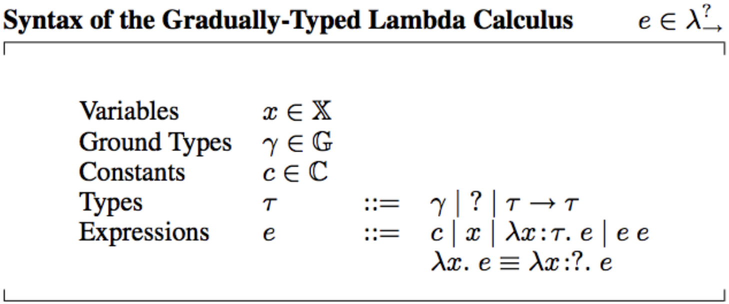
Gradual Typing

$\lambda^{?}_{\rightarrow}$

$\lambda^{\langle\tau\rangle}_{\rightarrow}$

Implementation

Syntax



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

Typing Rules

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)
$$\frac{\begin{array}{c}\Gamma \vdash_G e_1 : \tau \to \tau' \\ \Gamma \vdash_G e_2 : \tau_2 \qquad \tau_2 \sim \tau\end{array}}{\Gamma \vdash_G e_1\, e_2 : \tau'}$$

Run-time Semantics

- Requires a cast insertion translation from $\lambda^{?}_{\rightarrow}$ to an intermediate language
- Reason is that the type-checker requires each typable object to be "tagged" with its type to verify consistency

Topic

Gradual Typing

$\lambda^{?}_{\rightarrow}$

$\lambda^{\langle\tau\rangle}_{\rightarrow}$

Implementation

Syntax

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

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

Typing Rules

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

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

First-order example

Higher-order example

Relation to simply-typed λ-calculus

Theorem: If e $\in \lambda_{\rightarrow}$ then $\emptyset \vdash_{G} e : \tau \equiv \emptyset \vdash_{\rightarrow} e : \tau$

Proof

Type Safety

$\lambda^{?}_{\rightarrow}$ is type safe; if evaluation terminates, the result is either a value of an expected type or a cast error, but not a type error.

Theorem: If e $\in {\lambda_{\rightarrow}}^{?}$ with type τ can be converted to e' $\in {\lambda_{\rightarrow}}^{\langle ? \rangle}$ with type τ, then it will evaluate to result **r**, **r** a value, CastError, or KillError.

Topic

Gradual Typing

$\lambda^{?}_{\rightarrow}$

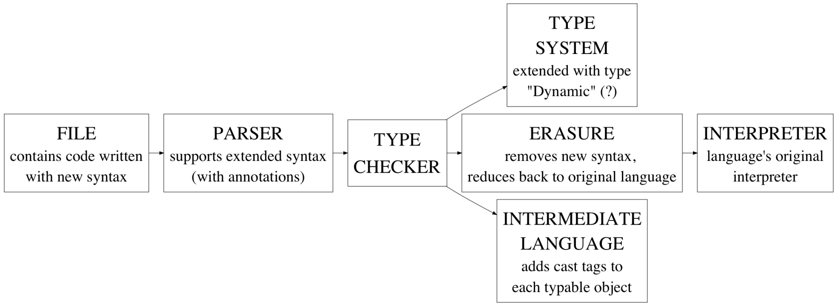
$\lambda^{\langle\tau\rangle}_{\rightarrow}$

Implementation

Requirements

1. Terms
2. Types
3. Typing Rules
4. Operational Semantics

General Workflow


FILE
contains code written with new syntax
PARSER
supports extended syntax (with annotations)
TYPE CHECKER
TYPE SYSTEM
extended with type "Dynamic" (?)
ERASURE
removes new syntax, reduces back to original language
INTERMEDIATE LANGUAGE
adds cast tags to each typable object
INTERPRETER
language's original interpreter

Implementation – Grammar

$$\gamma \quad ::= \quad \mathbb{N} \quad \mathbb{B} \quad \mathbb{C} \quad \mathbb{S}$$

$$\begin{array}{lcl} \sigma & ::= & ? \\ & | & \gamma \\ & | & (\sigma_1 * \sigma_2 * \cdots * \sigma_n) \\ & | & \mathsf{list}\,\sigma \\ & | & \sigma \rightarrow \tau \\ & | & \sigma \underset{n}{\rightarrow} \tau \end{array}$$

Implementation – Grammar (Prefix syntax)

$$\langle \textit{ground-type} \rangle \quad ::= \quad \texttt{number} \;\mid\; \texttt{boolean} \;\mid\; \texttt{char} \;\mid\; \texttt{string}$$

$$\begin{array}{rcl}
\langle \textit{type} \rangle & ::= & \texttt{any} \\
& \mid & \langle \textit{ground-type} \rangle \\
& \mid & (* \langle \textit{type} \rangle \langle \textit{type} \rangle \ldots \langle \textit{type} \rangle) \\
& \mid & \mathsf{list} \langle \textit{type} \rangle \\
& \mid & \rightarrow \; \langle \textit{type} \rangle \langle \textit{type} \rangle \\
& \mid & \rightarrow \textit{n} \; \langle \textit{type} \rangle \langle \textit{type} \rangle
\end{array}$$

Implementation – Grammar

$$
\begin{array}{lcl}
\langle \textit{expression} \rangle & ::= & \ldots \\
 & | & (\texttt{fn}\ (:\ \langle \textit{variable} \rangle \langle \textit{type} \rangle) \langle \textit{expression} \rangle) \\
 & | & (\texttt{fn}\ (:\ \langle \textit{variable} \rangle \langle \textit{type} \rangle)\ (:\ \langle \textit{type} \rangle) \langle \textit{expression} \rangle) \\
 & | & (\texttt{listof}\ (:\ \langle \textit{type} \rangle) \langle \textit{expression} \rangle *) \\
 & | & (\texttt{pair} \langle \textit{expression} \rangle \langle \textit{expression} \rangle) \\
 & | & (\texttt{defvar}\ (:\ \langle \textit{variable} \rangle \langle \textit{type} \rangle) \langle \textit{expression} \rangle) \\
 & | & (\texttt{defn}\ (:\ \langle \textit{variable} \rangle \langle \textit{type} \rangle)(\langle \textit{variable} \rangle *) \langle \textit{expression} \rangle)
\end{array}
$$

Implementation – Operational Semantics

- Scheme's operational semantics
- Macros enable type annotations
- Each macro simply performs erasure on 'itself'

Implementation – Operational Semantics

$$\frac{}{\texttt{(fn (: x s) M)} \rightarrow \texttt{(lambda (x) M)}}$$

$$\frac{}{\texttt{(listof (: s) m n ...)} \rightarrow \texttt{(list m n ...)}}$$

$$\frac{}{\texttt{(defvar (: x s) M)} \rightarrow \texttt{(define x M)}}$$

$$\frac{}{\texttt{(pair x y)} \rightarrow \texttt{(cons x y)}}$$

$$\frac{}{\texttt{(defn (: f s) (x y ...) M)} \rightarrow \texttt{(define (f x y ...) M)}}$$

Implementation – `listof` macro

```
(define-syntax listof
  (syntax-rules (:)
    ((_ (: type) e1 ...)
     (list e1 ...))))
```

Implementation – `fn` macro (1/2)

```
(define-syntax fn-erase
  (syntax-rules (:)
    ((_ (: v type))
     '(v))
    ((_ ((: v type) v2 ...))
     `(v ,@(fn-erase v2 ...)))))
```

Implementation – `fn` macro (2/2)

```
(define-syntax fn
  (syntax-rules (:)
    ((_ (: v type) (: return) body ...)
     (lambda (v) body ...))
    ((_ (: v type) body ...)
     (lambda (v) body ...))
    ((_ ((: v type) v2 ...) (: return) body ...)
     (fn ((: v type) v2 ...) body ...))
    ((_ ((: v type) v2 ...) body ...)
     (let ((env (the-environment)))
       (eval '(lambda ,(fn-erase ((: v type)) v2 ...))
       body ...) env))
    ((_ () body ...)
     (lambda () body ...))))
```

Implementation – `defn` macro

```
(define-syntax defn
  (syntax-rules (:)
    ((_ (: name type) (arg1 . args) body ...)
     (define (name arg1 . args) body ...))))
```

Implementation – Typing Rule – Application

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma' \qquad \sigma \sim \sigma' \qquad \sigma' \neq ? \text{ when } \sigma \neq ?}{\Gamma \vdash M\ N : \tau}$$

- Based on Typed Racket
- Otherwise `((fn (:  x any) (+ x 1)) #t)` would type check
- In *Gradual Typing for Functional Languages* this results in a *CastError* which is addressed in the operational semantics
- We don't have control over Scheme's operational semantics in our case
- If the function's domain type is not a `?` type, then the argument type cannot be `?`.

Justification – Typed Racket

```
((lambda ([x : Any]) (+ x 1)) #t)
```

```
Type Checker: type mismatch
  expected: Number
  given: Any
  in: x
```

Implementation – Typing Rules – Multiple Arity Functions

$$\frac{\Gamma \vdash M : \sigma_1 * \sigma_2 * \cdots * \sigma_n \to \tau \qquad \Gamma \vdash N_i^{i \in \{1,2,\ldots,n\}} : \sigma_i}{\Gamma \vdash (M\; N_1\; N_2 \ldots\; N_n) : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \underset{n}{\to} \tau \qquad \Gamma \vdash N_i^{i \in \{1,2,\ldots,n\}} : \sigma}{\Gamma \vdash (M\; N_1\; N_2 \ldots\; N_n) : \tau}$$

- $\underset{n}{\to}$ is not syntactic sugar for $\sigma * \sigma * \cdots * \sigma \to \tau$
- Helps deal with Scheme's multiple arity functions

Scheme's multiple arity functions

```
(+)           ; => 0
(+ 1)         ; => 1
(+ 1 2 3 4)   ; => 10

(*)           ; => 1
(* 1)         ; => 1
(* 1 2 3 4)   ; => 24

(>)           ; => #t
(> 1)         ; => #t
(> 1 2)       ; => #f
(> 3 2 1 0)   ; => #t

(: + (->n number number))
(: * (->n number number))
(: > (->n number boolean))
```

Implementation – Predefined types

```
(define predefined-types
  '((+ . (->n number number))
    (- . (->n number number))
    (* . (->n number number))
    (/ . (->n number number))
    (< . (->n number boolean))
    (> . (->n number boolean))
    (= . (->n number boolean))
    ...
    (null? . (-> (list any) boolean))
    (cons . (-> (* any (list any)) (list any)))
    (car . (-> (list any) any))
    (cdr . (-> (list any) (list any)))
    (map . (-> (* (-> any any) (list any)) (list any)))))
```

Implementation – Examples – `fn`

The function `t` takes an expression and an environment and returns the type of the expression

```
(t '(fn (: x number) (+ x 1)) predefined-types)
;; (-> number number)
```

```
(t '(fn (: x string) (+ x 1)) predefined-types)
;; TypeError: inconsistent argument types for +
```

Implementation – Examples – application

```
(t '((fn (: x number) (+ x 2 3)) 3) '())
;; number

(t '((fn (: x any) (+ x 1)) #t) predefined-types)
;; TypeError: inconsistent argument types for +

(t '((fn (: x any) (f x)) y) predefined-types)
;; any
;; f : (-> any any)
```

Implementation – Examples – listof

```
(t '(listof (: number) 1 2 3) '())
;; (list number)

(t '(listof (: any) 1 #t "H") '())
;; (list any)

(t '(listof (: number) 3 #t 5) '())
;; TypeError : Cast insertion error (: #t number)

(t '(listof (: number) (f 3) (f #t)) '())
;; TypeError : expected number got boolean for f

(t '(listof (: number) (f 3) (f #t))
   '((f . (-> any number))))
;; (list number)
```

Implementation – Examples – defn

```
(defn (: range (-> (* number number) (list number))) (x y)
  (if (> x y)
      (listof (: number))
      (cons x (range (+ x 1) y))))
```

A gradually typed interpreter

- As an example of a gradually typed program in our implementation, consider an interpreter for $\lambda^{\rightarrow,\mathbb{N}}$
- Example Programs:
  - `(int 3), (lam x (var x)), (app f x)`
  - `(app (lam x (var x)) (int 3))`
  - `(lam x (lam y (app (var x) (var y))))`
- The next few slides go through the implementation of this interpreter in our gradually typed system

A gradually typed interpreter

```
;; Types can be aliased
(type-alias 'Envr '(list (pair any any)))
(type-alias 'Expr '(list any))

;; typed functions
(defn (: value (-> Expr any)) (exp)
  (car (cdr exp)))
(defn (: type (-> Expr any)) (exp)
  (car exp))

;; untyped
(define operator cadr)
(define operand caddr)
(define param cadr)
(define body caddr)
```

A gradually typed interpreter – eval

```
;; Type Checked
(defn (: eval (-> (* Expr Envr) any)) (exp env)
  (if (eq? (type exp) 'var)
      (cdr (assoc (value exp) env))
      (if (eq? (type exp) 'int)
          (value exp)
          (if (eq? (type exp) 'app)
             (apply
               (eval (operator exp) env)
               (eval (operand exp) env))
             (if (eq? (type exp) 'lam)
                  (listof (: any)
                    (param exp) (body exp) env)
                  (listof (: any)))))))
```

A gradually typed interpreter – apply

```
;; Not type checked
(define (apply f arg)
  (pmatch f
    ((,x ,body ,env)
     (eval body (cons (cons x arg) env)))
    (else (error "expected closure in application"))))
```

Source + Demo

REPO HERE