The Design and Implementation of Typed Scheme

Mitch, Caelan, and David

The Paper

- “The Design and Implementation of Typed Scheme”
- Sam Tobin-Hochstadt and Matthias Felleisen
- Principles Of Programming Languages (POPL) Conference, 2008
- Introduced the first functioning model for a Gradual Typing system

The goal: add types to untyped languages!

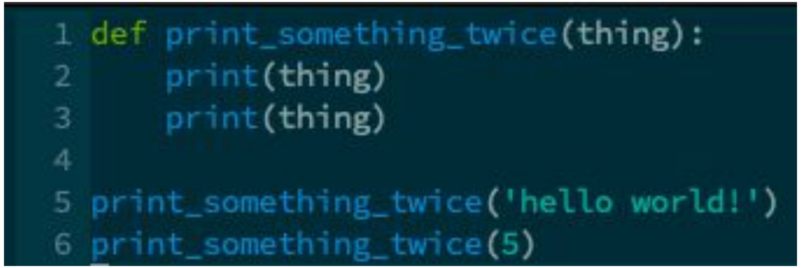
The Pros and Cons of Untyped Languages

Pros :

- Easy to get started
- Easy to configure
- Lots of flexibility

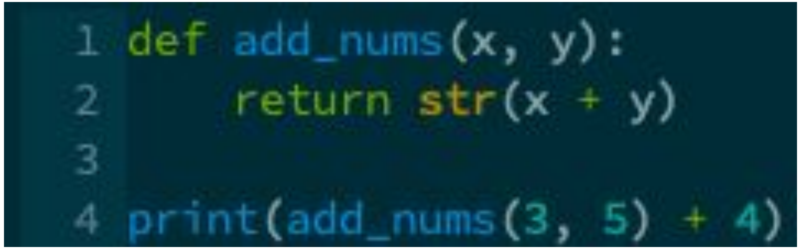
Cons:

- Harder to reason about
- Introduces lots of bugs that types prevent



```
def print_something_twice(thing):
    print(thing)
    print(thing)

print_something_twice('hello world!')
print_something_twice(5)
```



```
def add_nums(x, y):
    return str(x + y)

print(add_nums(3, 5) + 4)
```

What is Gradual Typing?

- A method for adding types back to code *gradually*
- Allows a programmer to write typed code that works together with untyped code

AND

- Convert existing untyped code to typed code

History of Gradual Typing

- Soft Typing (1990's)
- Migratory Typing (2006)
- Gradual Typing (2006)
- First full model for gradual typing (2008)

Interlanguage Migration: From Scripts to Programs

Sam Tobin-Hochstadt
Northeastern University
Boston, MA
samth@ccs.neu.edu

Matthias Felleisen
Northeastern University
Boston, MA
matthias@ccs.neu.edu

Gradual Typing for Functional Languages

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

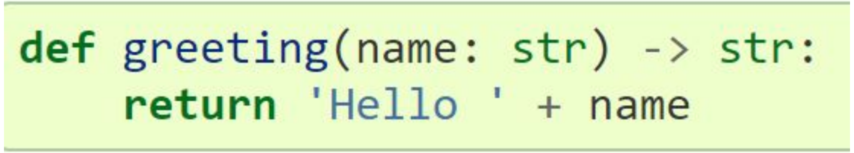
Walid Taha
Rice University
taha@rice.edu

Examples of Gradual Typing

Untyped Languages with Gradually Typed Equivalents:

- Javascript → Typescript
- Racket → Typed Racket
- PHP → Hack
- Python → mypy, Python3.8 type annotations, etc

Languages like Dart are being built gradually typed from the outset



```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Types of Gradual Typing

Micro:

- Add types to whatever part of the code you want, no matter how small
- Pros: Easy to add types
- Cons: tough on performance, is not very helpful with code guarantees

Macro:

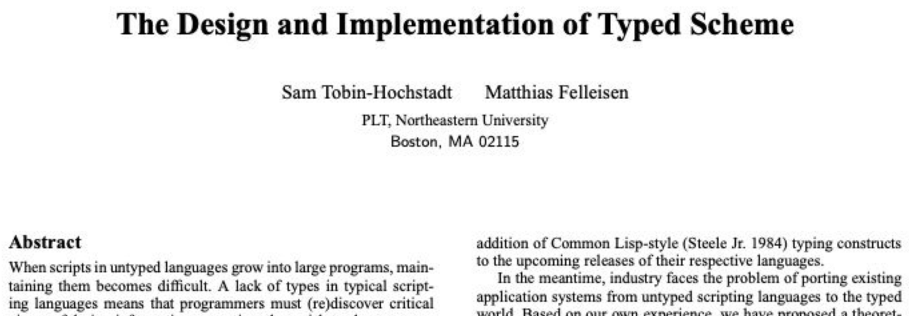
- Have (mostly) typed files/modules, and separate untyped files/modules
- Pros: better performance, stronger guarantees
- Cons: Higher upfront cost for migrating

Types of Type Checking

- Optional: cast values to the type they’re supposed to be and hope for the best
- Latent: Check the type of something when it’s used
- Natural: Check all types when a function is called

Methodology

- Give formal semantics for language (AKA define how the language works mathematically)
- Describe real language + how different operations work
- Describe examples + real code/migrations



The Design and Implementation of Typed Scheme

Sam Tobin-Hochstadt Matthias Felleisen
PLT, Northeastern University
Boston, MA 02115

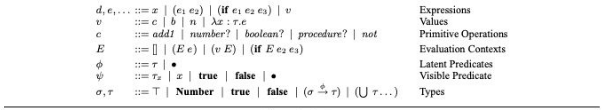
Abstract

When scripts in untyped languages grow into large programs, maintaining them becomes difficult. A lack of types in typical scripting languages means that programmers must (re)discover critical

addition of Common Lisp-style (Steele Jr. 1984) typing constructs to the upcoming releases of their respective languages.

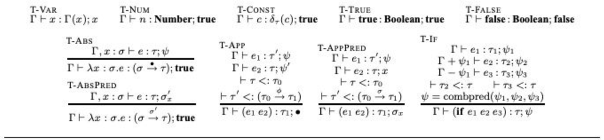
In the meantime, industry faces the problem of porting existing application systems from untyped scripting languages to the typed

Lambda Calculus



$d, e, \ldots$	$::= x \mid (e_1\ e_2) \mid (\mathbf{if}\ e_1\ e_2\ e_3) \mid v$	Expressions
v	$::= c \mid b \mid n \mid \lambda x:\tau.e$	Values
c	$::= \mathit{add1} \mid \mathit{number?} \mid \mathit{boolean?} \mid \mathit{procedure?} \mid \mathit{not}$	Primitive Operations
E	$::= [] \mid (E\ e) \mid (v\ E) \mid (\mathbf{if}\ E\ e_2\ e_3)$	Evaluation Contexts
ϕ	$::= \tau \mid \bullet$	Latent Predicates
ψ	$::= \tau_x \mid x \mid \mathbf{true} \mid \mathbf{false} \mid \bullet$	Visible Predicate
σ, τ	$::= \top \mid \mathbf{Number} \mid \mathbf{true} \mid \mathbf{false} \mid (\sigma \xrightarrow{\phi} \tau) \mid (\bigcup \tau \ldots)$	Types

Figure 1. Syntax



T-VAR $\Gamma \vdash x : \Gamma(x); x$

T-NUM $\Gamma \vdash n : \mathbf{Number}; \mathbf{true}$

T-CONST $\Gamma \vdash c : \delta_\tau(c); \mathbf{true}$

T-TRUE $\Gamma \vdash \mathbf{true} : \mathbf{Boolean}; \mathbf{true}$

T-FALSE $\Gamma \vdash \mathbf{false} : \mathbf{Boolean}; \mathbf{false}$

T-ABS
$$\frac{\Gamma, x:\sigma \vdash e : \tau; \psi}{\Gamma \vdash \lambda x:\sigma.e : (\sigma \xrightarrow{\bullet} \tau); \mathbf{true}}$$

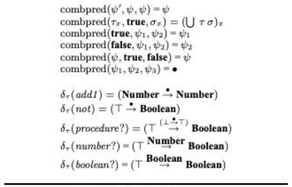
T-ABSPRED
$$\frac{\Gamma, x:\sigma \vdash e : \tau; \sigma'_x}{\Gamma \vdash \lambda x:\sigma.e : (\sigma \xrightarrow{\sigma'} \tau); \mathbf{true}}$$

T-APP
$$\frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash e_2 : \tau; \psi' \quad \vdash \tau <: \tau_0 \quad \vdash \tau' <: (\tau_0 \xrightarrow{\phi} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \bullet}$$

T-APPPRED
$$\frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash e_2 : \tau; x \quad \vdash \tau <: \tau_0 \quad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \sigma_x}$$

T-IF
$$\frac{\Gamma \vdash e_1 : \tau_1; \psi_1 \quad \Gamma + \psi_1 \vdash e_2 : \tau_2; \psi_2 \quad \Gamma - \psi_1 \vdash e_3 : \tau_3; \psi_3 \quad \vdash \tau_2 <: \tau \quad \vdash \tau_3 <: \tau \quad \psi = \mathrm{combpred}(\psi_1, \psi_2, \psi_3)}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \tau; \psi}$$

Figure 2. Primary Typing Rules



$\mathrm{combpred}(\psi', \psi, \psi) = \psi$

$\mathrm{combpred}(\tau_x, \mathbf{true}, \sigma_x) = (\bigcup\ \tau\ \sigma)_x$

$\mathrm{combpred}(\mathbf{true}, \psi_1, \psi_2) = \psi_1$

$\mathrm{combpred}(\mathbf{false}, \psi_1, \psi_2) = \psi_2$

$\mathrm{combpred}(\psi, \mathbf{true}, \mathbf{false}) = \psi$

$\mathrm{combpred}(\psi_1, \psi_2, \psi_3) = \bullet$

$\delta_\tau(\mathit{add1}) = (\mathbf{Number} \xrightarrow{\bullet} \mathbf{Number})$

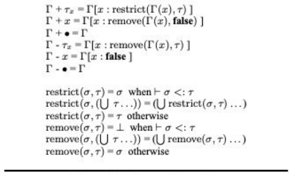
$\delta_\tau(\mathit{not}) = (\top \xrightarrow{\bullet} \mathbf{Boolean})$

$\delta_\tau(\mathit{procedure?}) = (\top \xrightarrow{(\bot \xrightarrow{\bullet} \top)} \mathbf{Boolean})$

$\delta_\tau(\mathit{number?}) = (\top \xrightarrow{\mathbf{Number}} \mathbf{Boolean})$

$\delta_\tau(\mathit{boolean?}) = (\top \xrightarrow{\mathbf{Boolean}} \mathbf{Boolean})$

Figure 3. Auxiliary Operations



$\Gamma + \tau_x = \Gamma[x : \mathrm{restrict}(\Gamma(x), \tau)\ ]$

$\Gamma + x = \Gamma[x : \mathrm{remove}(\Gamma(x), \mathbf{false})\ ]$

$\Gamma + \bullet = \Gamma$

$\Gamma - \tau_x = \Gamma[x : \mathrm{remove}(\Gamma(x), \tau)\ ]$

$\Gamma - x = \Gamma[x : \mathbf{false}\ ]$

$\Gamma - \bullet = \Gamma$

$\mathrm{restrict}(\sigma, \tau) = \sigma$ when $\vdash \sigma <: \tau$

$\mathrm{restrict}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup\ \mathrm{restrict}(\sigma, \tau) \ldots)$

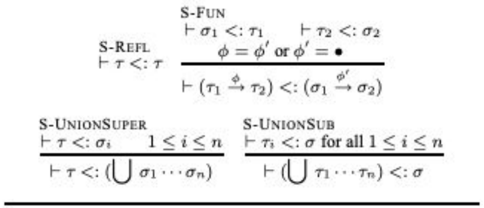
$\mathrm{restrict}(\sigma, \tau) = \tau$ otherwise

$\mathrm{remove}(\sigma, \tau) = \bot$ when $\vdash \sigma <: \tau$

$\mathrm{remove}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup\ \mathrm{remove}(\sigma, \tau) \ldots)$

$\mathrm{remove}(\sigma, \tau) = \sigma$ otherwise

Figure 4. Environment Operations



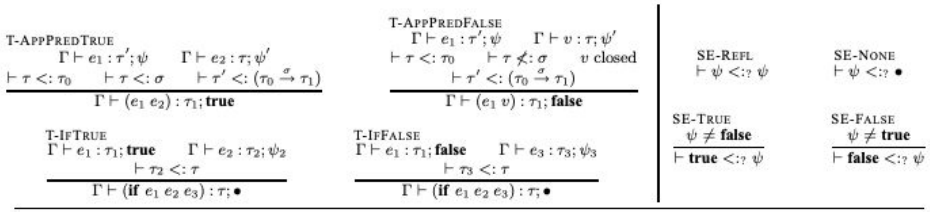
S-REFL $\vdash \tau <: \tau$

S-FUN
$$\frac{\vdash \sigma_1 <: \tau_1 \quad \vdash \tau_2 <: \sigma_2 \quad \phi = \phi' \text{ or } \phi' = \bullet}{\vdash (\tau_1 \xrightarrow{\phi} \tau_2) <: (\sigma_1 \xrightarrow{\phi'} \sigma_2)}$$

S-UNIONSUPER
$$\frac{\vdash \tau <: \sigma_i \quad 1 \le i \le n}{\vdash \tau <: (\bigcup\ \sigma_1 \cdots \sigma_n)}$$

S-UNIONSUB
$$\frac{\vdash \tau_i <: \sigma \text{ for all } 1 \le i \le n}{\vdash (\bigcup\ \tau_1 \cdots \tau_n) <: \sigma}$$

Figure 5. Subtyping Relation



T-APPPREDTRUE
$$\frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash e_2 : \tau; \psi' \quad \vdash \tau <: \tau_0 \quad \vdash \tau <: \sigma \quad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \mathbf{true}}$$

T-APPPREDFALSE
$$\frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash v : \tau; \psi' \quad \vdash \tau <: \tau_0 \quad \vdash \tau \not<: \sigma \quad v \text{ closed} \quad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ v) : \tau_1; \mathbf{false}}$$

T-IFTRUE
$$\frac{\Gamma \vdash e_1 : \tau_1; \mathbf{true} \quad \Gamma \vdash e_2 : \tau_2; \psi_2 \quad \vdash \tau_2 <: \tau}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \tau; \bullet}$$

T-IFFALSE
$$\frac{\Gamma \vdash e_1 : \tau_1; \mathbf{false} \quad \Gamma \vdash e_3 : \tau_3; \psi_3 \quad \vdash \tau_3 <: \tau}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \tau; \bullet}$$

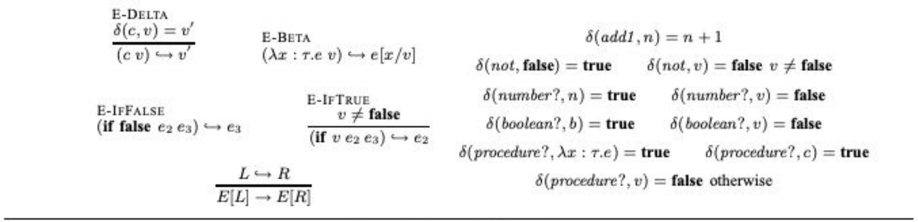
SE-REFL $\vdash \psi <:_? \psi$

SE-NONE $\vdash \psi <:_? \bullet$

SE-TRUE
$$\frac{\psi \neq \mathbf{false}}{\vdash \mathbf{true} <:_? \psi}$$

SE-FALSE
$$\frac{\psi \neq \mathbf{true}}{\vdash \mathbf{false} <:_? \psi}$$

Figure 6. Auxiliary Typing Rules



E-DELTA
$$\frac{\delta(c, v) = v'}{(c\ v) \hookrightarrow v'}$$

E-BETA $(\lambda x:\tau.e\ v) \hookrightarrow e[x/v]$

E-IFFALSE $(\mathbf{if}\ \mathbf{false}\ e_2\ e_3) \hookrightarrow e_3$

E-IFTRUE
$$\frac{v \neq \mathbf{false}}{(\mathbf{if}\ v\ e_2\ e_3) \hookrightarrow e_2}$$

$$\frac{L \hookrightarrow R}{E[L] \rightarrow E[R]}$$

$\delta(\mathit{add1}, n) = n + 1$

$\delta(\mathit{not}, \mathbf{false}) = \mathbf{true}$ $\quad \delta(\mathit{not}, v) = \mathbf{false}\ \ v \neq \mathbf{false}$

$\delta(\mathit{number?}, n) = \mathbf{true}$ $\quad \delta(\mathit{number?}, v) = \mathbf{false}$

$\delta(\mathit{boolean?}, b) = \mathbf{true}$ $\quad \delta(\mathit{boolean?}, v) = \mathbf{false}$

$\delta(\mathit{procedure?}, \lambda x:\tau.e) = \mathbf{true}$ $\quad \delta(\mathit{procedure?}, c) = \mathbf{true}$

$\delta(\mathit{procedure?}, v) = \mathbf{false}$ otherwise

Figure 7. Operational Semantics

Examples: Typed ISL/PLT Scheme

```
;; add two numbers
;; Number Number -> Number
(define (add x y)
        (+ x y))
```

```
;; add two numbers
(define (add [x: Number] [y: Number]): Number
        (+ x y))
```

Examples: Typed ISL/PLT Scheme

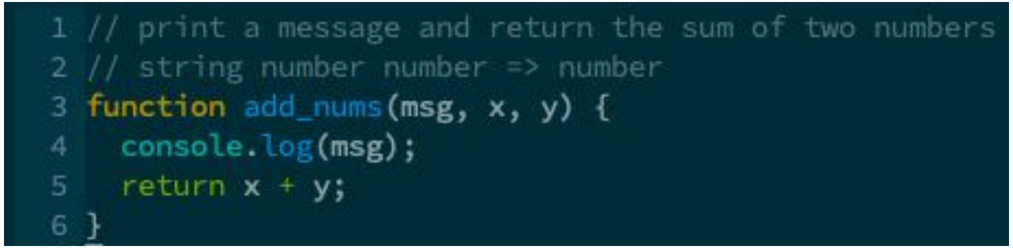
```
;; data definition: a Complex is one of:
;; - a Number or
;; - (cons Number Number)

;; Complex → Number
(define (creal x)
  (cond [(number? x) x]
        [else (car x)]))
```

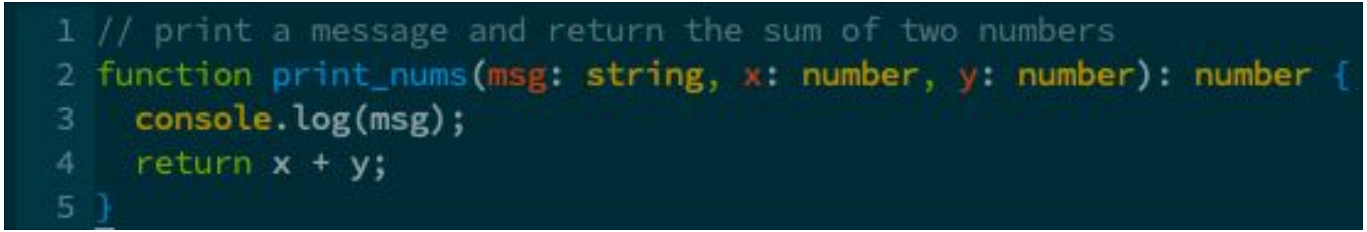
```
(define-type-alias Cplx (∪ Number (cons Number Number)))

(define: (creal [x : Cplx]) : Number
  (cond [(number? x) x]
        [else (car x)]))
```

Examples: Typescript

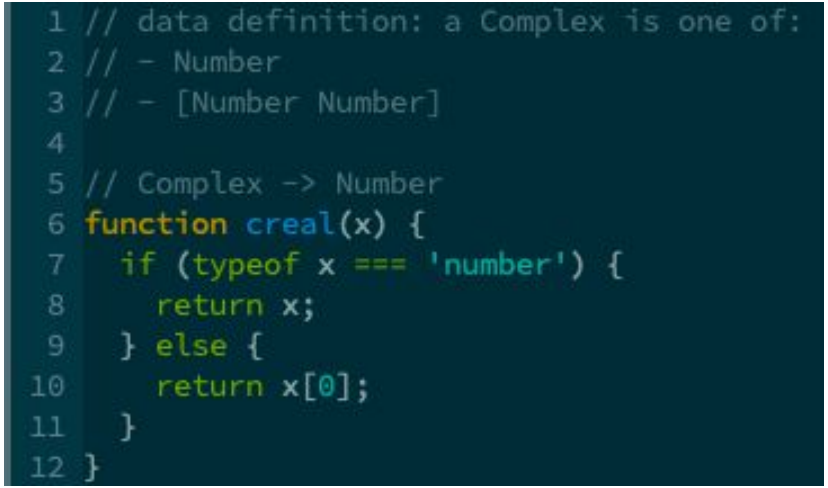


```
// print a message and return the sum of two numbers
// string number number => number
function add_nums(msg, x, y) {
  console.log(msg);
  return x + y;
}
```



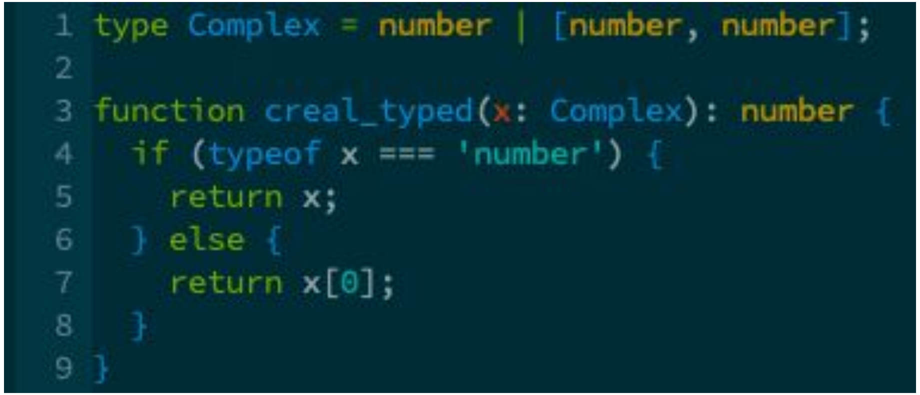
```
// print a message and return the sum of two numbers
function print_nums(msg: string, x: number, y: number): number {
  console.log(msg);
  return x + y;
}
```

Examples: Typescript



```
// data definition: a Complex is one of:
// - Number
// - [Number Number]

// Complex -> Number
function creal(x) {
  if (typeof x === 'number') {
    return x;
  } else {
    return x[0];
  }
}
```



```
type Complex = number | [number, number];

function creal_typed(x: Complex): number {
  if (typeof x === 'number') {
    return x;
  } else {
    return x[0];
  }
}
```

Additional Features

- ‘Local’ Type Inference
- Type checking on structs & lists/list operations (map, filter, etc)

```
(define (m [z : Number]): Number
     (define x 3)
     (define y (* x x))
     (- y 1))
```

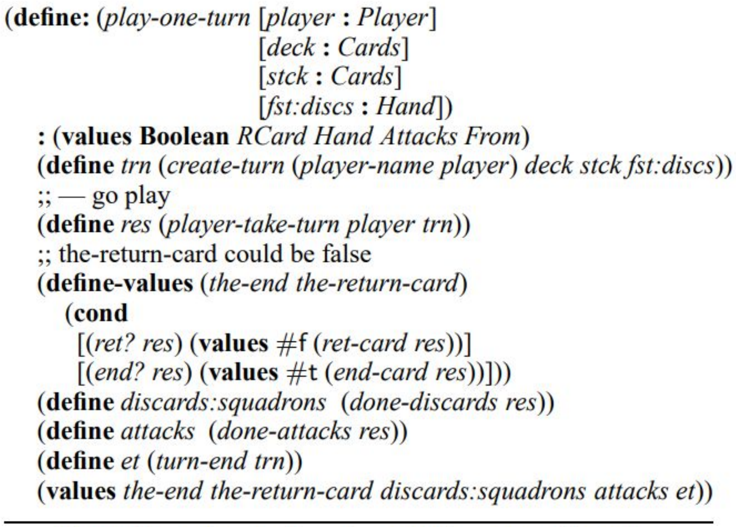
Proof of Concept

To prove the effectiveness of the model, programs were ported into the gradual typing system.

- Code from How To Design Programs (HtDP)
- Small libraries in the Racket codebase
- An online game + a checkbook system (written and ported by an undergrad!)

Major challenges:

- Generic ('polymorphic') Types
- Classes/Objects
- Racket Macros

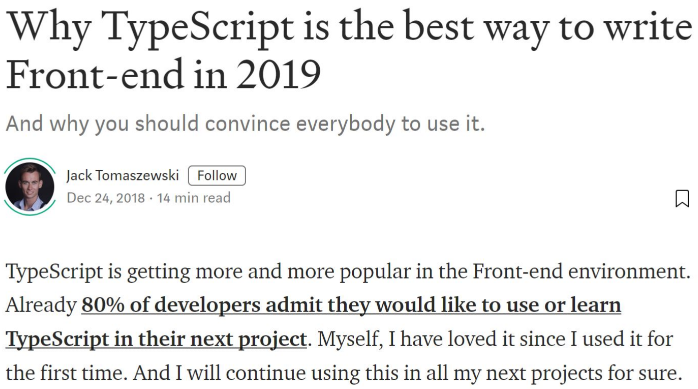


```
(define: (play-one-turn [player : Player]
                        [deck : Cards]
                        [stck : Cards]
                        [fst:discs : Hand])
  : (values Boolean RCard Hand Attacks From)
  (define trn (create-turn (player-name player) deck stck fst:discs))
  ;; — go play
  (define res (player-take-turn player trn))
  ;; the-return-card could be false
  (define-values (the-end the-return-card)
    (cond
      [(ret? res) (values #f (ret-card res))]
      [(end? res) (values #t (end-card res))]))
  (define discards:squadrons  (done-discards res))
  (define attacks  (done-attacks res))
  (define et (turn-end trn))
  (values the-end the-return-card discards:squadrons attacks et))
```

Figure 9. A Excerpt from the Squadron Scramble Game

Impact of Paper & Future of Gradual Typing

- First complete model for gradual typing
- Significant influence on gradually-typed languages in industry
  - Particularly Typescript
- Inspired further research into gradual typing models in other programming languages
  - Ruby
  - Javascript
- Gradual Typing for specific challenges
  - Generic Typing



Why TypeScript is the best way to write Front-end in 2019

And why you should convince everybody to use it.

Jack Tomaszewski Follow
Dec 24, 2018 · 14 min read

TypeScript is getting more and more popular in the Front-end environment. Already **80% of developers admit they would like to use or learn TypeScript in their next project**. Myself, I have loved it since I used it for the first time. And I will continue using this in all my next projects for sure.

Regrets

Matthias Felleisen had two regrets with his paper:

- Got the name wrong—he published first, but Siek came up with the Gradual Typing name
- Sam spent 6 months trying to use an automatic prover to prove properties about the language, which ultimately failed
    - Delayed paper by half a year

Want to do Research?

Northeastern Programming Research Laboratory (PRL)

- Amal Ahmed working on Gradual Types
- Other significant research areas:
    - Type Systems
    - Compilers
    - Domain-Specific Languages
    - … and much, much more!

Citations

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. SIGPLAN Not. 43, 1 (January 2008), 395–406. DOI:https://doi.org/10.1145/1328897.1328486

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. InScheme and Functional Programming Workshop, University of Chicago,Technical Report TR-2006-06, pages 81–92, September 2006.

Tobin-Hochstadt, Sam, and Matthias Felleisen. "Interlanguage migration: from scripts to programs." *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006.

New, Max S., Dustin Jamner, and Amal Ahmed. "Graduality and parametricity: together again for the first time." *Proceedings of the ACM on Programming Languages* 4.POPL (2019): 1-32.