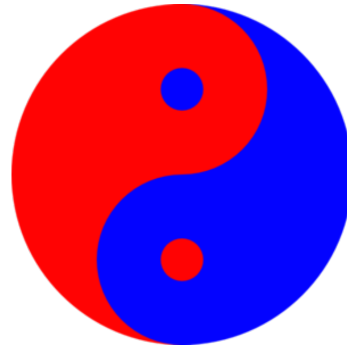
We are all poor schmucks: On the value of gradual types

Philip Wadler
University of Edinburgh

PADL, New Orleans, Tuesday 21 January 2020



Part I

From untyped to typed

An untyped program

$$\ulcorner \texttt{let}$$

$$f = \lambda y.\, y + \texttt{1}$$

$$h = \lambda g.\, g\ \texttt{3})$$

$$\texttt{in}$$

$$h\ f \urcorner$$

$$\longrightarrow$$

$$\ulcorner \texttt{4} \urcorner$$

A typed program

$$\texttt{let}$$

$$f = \lambda y : \texttt{Int}.\, y + 1$$

$$h = \lambda g : \texttt{Int} \rightarrow \texttt{Int}.\, g\ 3$$

$$\texttt{in}$$

$$h\ f$$

$$\longrightarrow$$

$$4 : \texttt{Int}$$

An untyped term in a typed context

$$\texttt{let}$$

$$f = \lceil \lambda y.\, y + \texttt{1} \rceil : \star \overset{p}{\Rightarrow} \texttt{Int} \to \texttt{Int}$$

$$h = \lambda g : \texttt{Int} \to \texttt{Int}.\, g\ \texttt{3}$$

$$\texttt{in}$$

$$h\ f$$

$$\longrightarrow$$

$$\texttt{4} : \texttt{Int}$$

An untyped term in a typed context

$$\begin{array}{l}\texttt{let} \\ \quad f = \lceil \lambda y.\, y > \texttt{1} \rceil : \star \overset{p}{\Longrightarrow} \texttt{Int} \to \texttt{Int} \\ \quad h = \lambda g : \texttt{Int} \to \texttt{Int}.\, g\ \texttt{3} \\ \texttt{in} \\ \quad h\ f \end{array}$$

$$\longrightarrow$$

$$\texttt{blame}\ p$$

Positive: blame the term contained in the cast

A typed term in an untyped context

$$
\begin{array}{l}
\texttt{let} \\
\quad f = (\lambda y : \texttt{Int}.\, y + \texttt{1}) : \texttt{Int} \to \texttt{Int} \overset{p}{\Rightarrow} \star \\
\quad h = \lceil \lambda g.\, g\ \texttt{3} \rceil \\
\texttt{in} \\
\quad \lceil h\ f \rceil \\
\longrightarrow \\
\lceil 4 \rceil
\end{array}
$$

A typed term in an untyped context

$$\texttt{let}$$

$$f = (\lambda y : \texttt{Int}.\, y + 1) : \texttt{Int} \to \texttt{Int} \stackrel{p}{\Longrightarrow} \star$$

$$h = \lceil \lambda g.\, g\ \texttt{true} \rceil$$

$$\texttt{in}$$

$$\lceil h\ f \rceil$$

$$\longrightarrow$$

$$\texttt{blame}\ \bar{p}$$

Negative: blame the context containg the cast

Part II

Blame safety

$<:$ $\quad <:^{+}$ $\quad <:^{-}$ $\quad <:_{n}$

The Wrap rule

$$(V : A \to B \overset{p}{\Longrightarrow} A' \to B')\, W \quad \longrightarrow \quad (V\ (W : A' \overset{\overline{p}}{\Longrightarrow} A)) : B \overset{p}{\Longrightarrow} B')$$

The Tangram Lemma

Lemma 1 (Tangram)

1. $A <: B$ *iff* $A <:^{+} B$ *and* $A <:^{-} B$.
2. $A <:_{n} B$ *iff* $A <:^{+} B$ *and* $B <:^{-} A$.

Well-typed programs can't be blamed

$$
\begin{array}{l}
\texttt{let} \\
\quad f = \lceil \lambda y.\, y + \texttt{1} \rceil : \star \stackrel{p}{\Longrightarrow} \texttt{Int} \to \texttt{Int} \\
\quad h = \lambda g : \texttt{Int} \to \texttt{Int}.\, g\ \texttt{3} \\
\texttt{in} \\
\quad h\ f
\end{array}
$$

$$\longrightarrow$$

$$\texttt{4} : \texttt{Int}$$

$$\texttt{Int} \to \texttt{Int} <:_n \star$$

implies

$$\star <:^- \texttt{Int} \to \texttt{Int}$$

Well-typed programs can't be blamed

$$
\begin{array}{l}
\texttt{let} \\
\quad f = (\lambda y : \texttt{Int}.\, y + \texttt{1}) : \texttt{Int} \to \texttt{Int} \stackrel{p}{\Longrightarrow} \star \\
\quad h = \lceil \lambda g.\, g\ \texttt{3} \rceil \\
\texttt{in} \\
\quad \lceil h\ f \rceil
\end{array}
$$

$$\longrightarrow$$

$$\lceil 4 \rceil$$

$$\texttt{Int} \to \texttt{Int} <:_n \star$$

implies

$$\texttt{Int} \to \texttt{Int} <:^+ \star$$

Well-typed programs can't go wrong

$$\texttt{let}$$

$$f = (\lambda y : \star.\, (y : \star \overset{q}{\Rightarrow} \texttt{Int}) + \texttt{1}) : \star \to \texttt{Int} \overset{p}{\Rightarrow} \texttt{Int} \to \star$$

$$h = \lambda g : \texttt{Int} \to \star.\, g\ \texttt{3}$$

$$\texttt{in}$$

$$h\ f$$

$$\longrightarrow$$

$$\lceil 4 \rceil$$

$$\star \to \texttt{Int} <: \texttt{Int} \to \star$$

implies

$$\star \to \texttt{Int} <:^{+} \texttt{Int} \to \star \text{ and } \star \to \texttt{Int} <:^{-} \texttt{Int} \to \star$$

Part III

Practice and theory

Enforcing declared types

$$\texttt{let}$$
$$\quad f = \lceil \lambda y : \texttt{Int}.\, y + 1 \rceil : \star \stackrel{p}{\Longrightarrow} \texttt{Int} \to \texttt{Int} \stackrel{p}{\Longrightarrow} \star$$
$$\quad h = \lceil \lambda g.\, g\ 3 \rceil$$
$$\texttt{in}$$
$$\quad \lceil h\ f \rceil$$
$$\longrightarrow$$
$$\lceil 4 \rceil$$

A wide-spectrum language

1. Dynamic types, as in Racket, Python, and JavaScript.
2. Polymorphic types, as in ML, Haskell, and F#.
3. Refinement types, as in Dependent ML and F7.
4. Dependent types, as in Coq, Agda, and F$\star$.

Part IV

Conclusion

Milner (1978):

Well-typed programs can’t go wrong.

Felleisen and Wright (1994); Harper (2002):

Well-typed programs don’t get stuck.

Wadler and Findler (2008):

Well-typed programs can’t be blamed.

- Findler and Wadler,
  Well-typed programs can't be blamed,
  ESOP 2009.

- Siek, Thiemann, and Wadler,
  Blame and coercion: Together again for the first time,
  PLDI 2015.

- Wadler,
  Abstract Data Types without the Types,
  Turner Festschrift, JUCS, 2017.

- Igarashi, Thiemann, Vasconcelos, and Wadler,
  Gradual session types,
  ICFP 2017.

- Zalewski, McKinna, Morris, Wadler,
  λdB: Blame tracking at higher fidelity,
  WGT 2020.

- Findler and Felleisen,
  Contracts for higher-order functions,
  ICFP 2002. [Contracts]

- Siek and Taha,
  Gradual typing for functional programming,
  Scheme Workshop, 2006. [Gradual types]

- Siek, Vitousek, Cimini, and Boyland,
  Refined Criteria for Gradual Typing,
  SNAPL, 2015. [Gradual guarantee]

- Garcia, Clark, and Tanter,
  Abstracting gradual typing,
  POPL 2016. [AGT]

- Eremondi, Tanter, Garcia,
  Approximate normalisation for gradual dependent types,
  ICFP 2019. [Dependent types need gradual values]

schmuck noun

\ ˈshmək \

plural **schmucks**

Definition of *schmuck*

slang

: a stupid, foolish, or unlikeable person **:** JERK sense 1b

// Do not be the poor *schmuck* who runs out of gas and is stranded when a natural disaster is about to hit.

— Ryan Carlyle

// ... his realization that he's ... like the rest of us, just an average *schmuck* who makes mistakes and tries to fix them.

— Maureen Ryan

// ... cursing under your breath as some other *schmuck* wins all the Bingo prizes ...

— *PortlandMercury.com*

// ... the phenomenon known as road rage, in which aggressive *schmucks* attack other people who get in the way of their four-by-fours.

— Stanley Bing

// In the very early days, we used to do all sorts of stuff that no one would have suspected of us, so that when we did get to the level of "The Ed Sullivan Show," we were real and not just some little *schmucks* from out of town.

— Paul McCartney