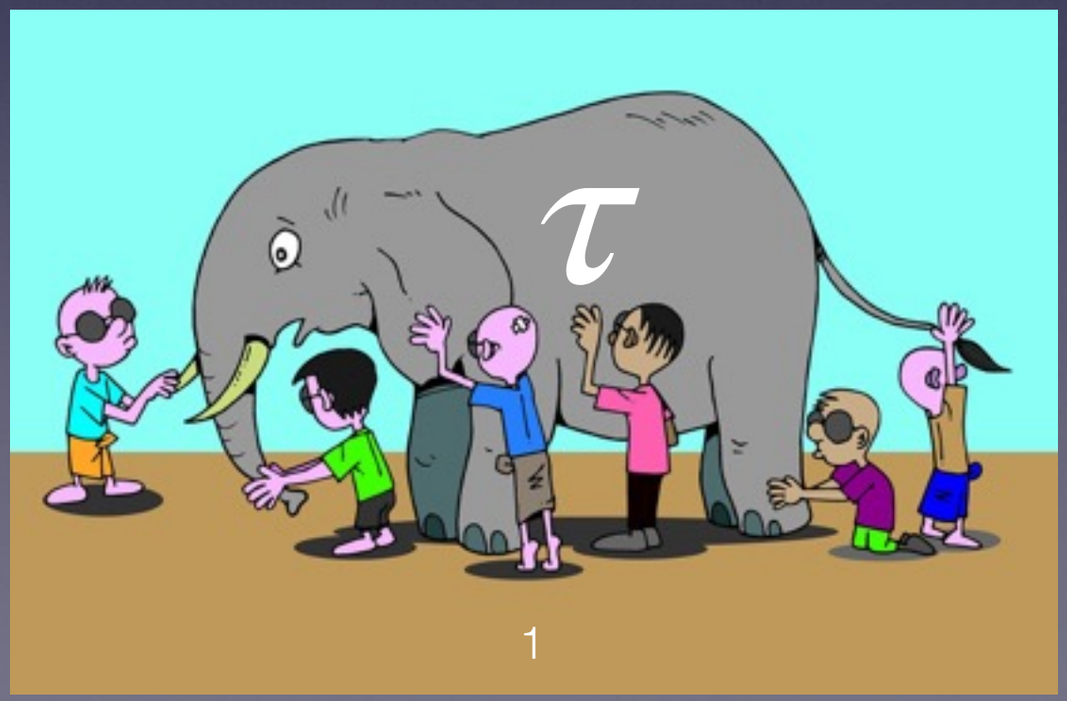
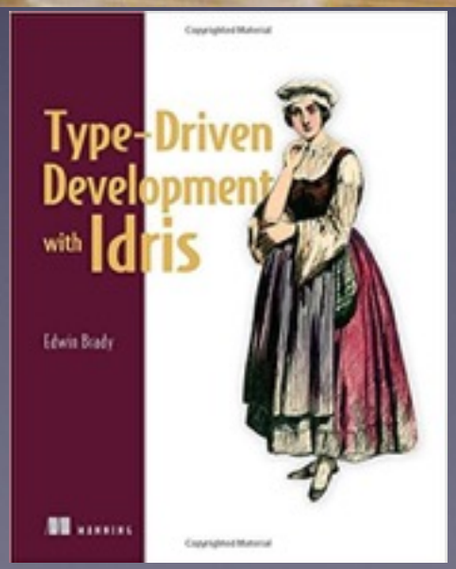
What type of thing is a type?

Ronald Garcia
University of British Columbia



Type Discipline (two practitioner views)

Type Discipline (two practitioner views)




```
(* ... in a function later in the file ... *)
        add vList1 s
```

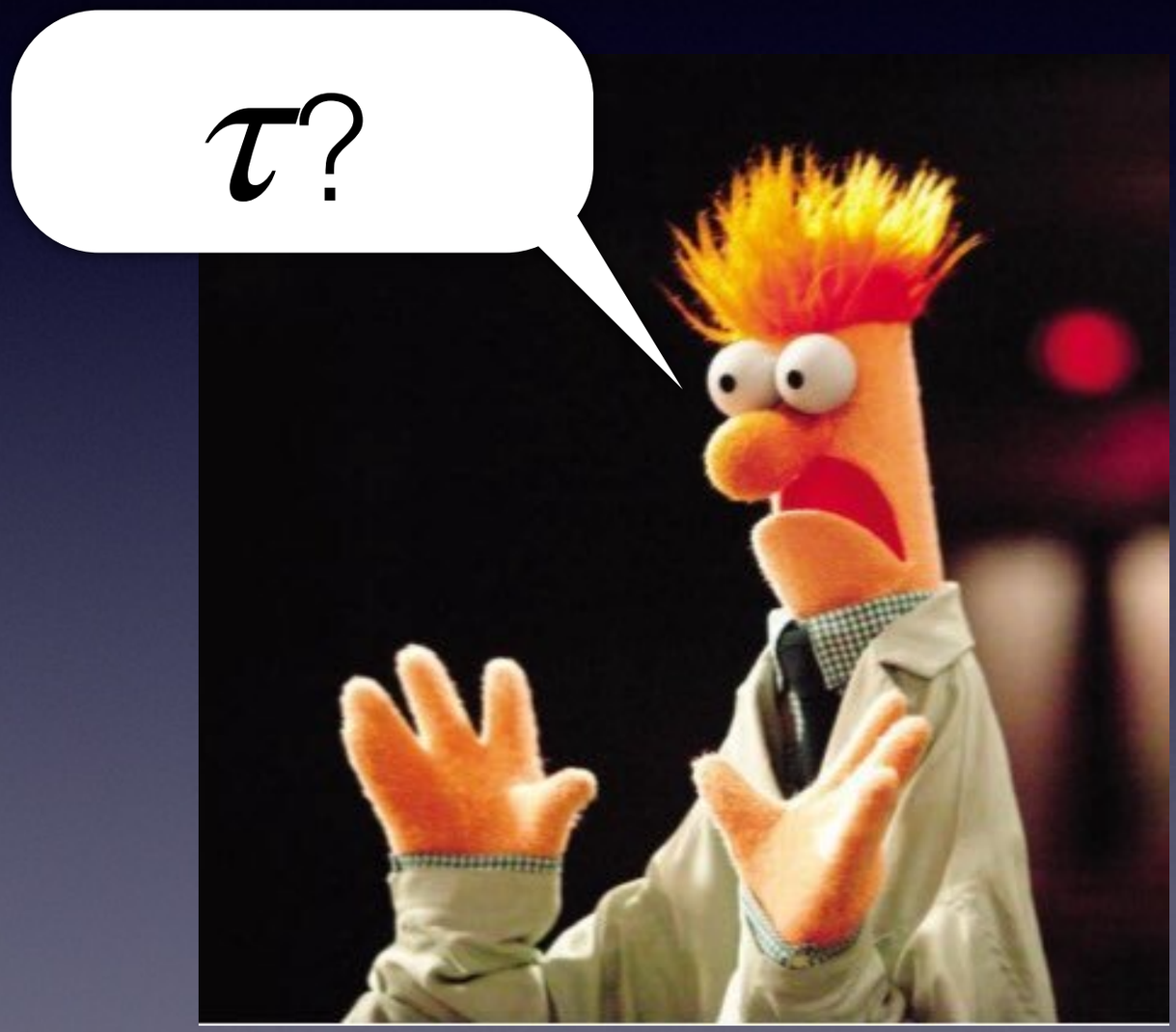
Type-checker:

```
The expression s has type string
but is here used with type string list list
```

Our approach:

```
Try replacing add vList1 s
with add s vList1
...
```

This talk



A theoretician's view

About Me

About Me

Dynamic Typing

Static Typing

About Me

Dynamic Typing

Static Typing

A Chronology (4 decades)

References:

Types Are Not Sets
by James H. Morris (1973)

A Theory of Type Polymorphism in Programming
by Robin Milner (1978)

Types, Abstraction, and Parametric Polymorphism
by John Reynolds (1983)

Co-induction in Relational Semantics
by Robin Milner and Mads Tofte (1990)

Typing First-class Continuations in ML
by Bruce Duba, Robert Harper, and David MacQueen (1991)

A Syntactic Approach to Type Soundness
by Andrew Wright and Matthias Felleisen (1994)

Semantics of Types for Mutable State
by Amal Ahmed (2004)

A Chronology (4 decades)

References:

Types Are Not Sets
by James H. Morris (1973)

A Theory of Type Polymorphism in Programming
by Robin Milner (1978)

Types, Abstraction, and Parametric Polymorphism
by John Reynolds (1983)

Co-induction in Relational Semantics
by Robin Milner and Mads Tofte (1990)

Typing First-class Continuations in ML
by Bruce Duba, Robert Harper, and David MacQueen (1991)

A Syntactic Approach to Type Soundness
by Andrew Wright and Matthias Felleisen (1994)

Semantics of Types for Mutable State
by Amal Ahmed (2004)

Themes

Syntax

Themes

Semantics

Themes

Pragmatics

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

LCF Theorem Prover
ML Programming Language
Calculus of Communicating Systems
Pi Calculus

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Turing Award (1991)

LCF Theorem Prover
ML Programming Language
Calculus of Communicating Systems
Pi Calculus

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Turing Award (1991)

Did not have a PhD!

LCF Theorem Prover
ML Programming Language
Calculus of Communicating Systems
Pi Calculus

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Turing Award (1991)

Did not have a PhD!

"The Hindley/Milner Type Inference Paper"

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Turing Award (1991)

Did not have a PhD!

"The Hindley/Milner Type Inference Paper"

A Conceptual Blueprint for Analyzing Types!

Look Ma, a Language!

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta.$

Look Ma, a Language!

Syntax

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta.$

Look Ma, a Language!

Syntax

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

Basically a Data Type

```
(require plai/datatype) ;; for define-type and type-case

(define-type Exp
  [var (x symbol?)]
  [app (e Exp?) (e^ Exp?)]
  [if-then-else (e Exp?) (e^ Exp?) (e^^ Exp?)]
  [lam (x symbol?) (e Exp?)]
  [fix (x symbol?) (e Exp?)]
  [let-in (x symbol?) (e Exp?) (e^ Exp?)])
```

Look Ma, a Language!

Syntax

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

Basically a Data Type

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta.$

Look Ma, a Language!

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

Semantics

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta.$

Look Ma, a Language!

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \quad (i = 1, 2)$.

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{ wrong}$$
where v_i is $\mathscr{E}[\![e_i]\!]\eta \quad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$
where $v_1 = \mathscr{E}[\![e_1]\!]\eta$.

Semantics

Closely Resembles an Interpreter(*)

$\mathscr{E}[\![e]\!]\eta$

```
((eval e) env)
```

$$\mathscr{E} \in \text{Exp} \to \text{Env} \to V,$$

Look Ma, a Language!

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \mathbin{\mathsf{E}} F \to (v_2 \mathbin{\mathsf{E}} W \to \text{wrong}, (v_1 \mid F)v_2), \text{wrong}$$

where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \mathbin{\mathsf{E}} B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{wrong}$$

where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \mathbin{\mathsf{E}} W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$

where $v_1 = \mathscr{E}[\![e_1]\!]\eta$.

It's a Dynamic Language!

Look Ma, a Language!

$$e ::= x \mid (ee') \mid \textit{if } e \textit{ then } e' \textit{ else } e'' \mid \lambda x \cdot e \mid \textit{fix } x \cdot e \mid \textit{let } x = e \textit{ in } e'.$$

$$\mathscr{E}[\![x]\!]\eta = \eta[\![x]\!]$$

$$\mathscr{E}[\![(e_1 e_2)]\!]\eta = v_1 \,\mathsf{E}\, F \to (v_2 \,\mathsf{E}\, W \to \text{wrong}, (v_1 \mid F)v_2), \text{wrong}$$

where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2).$

$$\mathscr{E}[\![\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3]\!]\eta = v_1 \,\mathsf{E}\, B_0 \to (v_1 \mid B_0 \to v_2, v_3), \text{wrong}$$

where v_i is $\mathscr{E}[\![e_i]\!]\eta \qquad (i = 1, 2, 3)$

$$\mathscr{E}[\![\lambda x \cdot e]\!]\eta = (\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\}) \text{ in } V$$

$$\mathscr{E}[\![\textit{fix } x \cdot e]\!]\eta = Y(\lambda v \cdot \mathscr{E}[\![e]\!]\,\eta\{v/x\})$$

$$\mathscr{E}[\![\textit{let } x = e_1 \textit{ in } e_2]\!]\eta = v_1 \,\mathsf{E}\, W \to \text{wrong}, \mathscr{E}[\![e_2]\!]\,\eta\{v_1/x\}$$

where $v_1 = \mathscr{E}[\![e_1]\!]\eta$

It's a Dynamic Language!
(all closed programs run)

Look Pa, Types!

Syntax

The syntax of types is as follows.

(1) ι_0 , ι_1 ,... are (basic) types; one for each B_i .

(2) There is a denumerable set of type variables, which are types. We use α, β, γ,... to range over type variables.

(3) If ρ and σ are types, so is $\rho \rightarrow \sigma$.

Basically
a
Data Type

Look Pa, Types!

Syntax

The syntax of types is as follows.

(1) $\iota_0, \iota_1, \ldots$ are (basic) types; one for each B_i.

(2) There is a denumerable set of type variables, which are types. We use $\alpha, \beta, \gamma, \ldots$ to range over type variables.

(3) If ρ and σ are types, so is $\rho \to \sigma$.

Basically a Data Type

$$\mu, \nu \quad ::= \quad \iota_1 \mid \iota_2 \mid \cdots \mid \mu \to \nu$$

$$\rho, \sigma \quad ::= \alpha \mid \iota_1 \mid \iota_2 \mid \cdots \mid \rho \to \sigma$$

Look Pa, Types!

Syntax

The syntax of types is as follows.

(1) $\iota_0, \iota_1, \ldots$ are (basic) types; one for each B_i.

(2) There is a denumerable set of type variables, which are types. We use $\alpha, \beta, \gamma, \ldots$ to range over type variables.

(3) If ρ and σ are types, so is $\rho \to \sigma$.

Basically a Data Type

$$\mu, \nu \quad ::= \quad \iota_1 \mid \iota_2 \mid \cdots \mid \mu \to \nu \quad \text{“monotypes”}$$

$$\rho, \sigma \quad ::= \alpha \mid \iota_1 \mid \iota_2 \mid \cdots \mid \rho \to \sigma$$

$$\text{INT}$$

$$\text{INT} \to \text{BOOL}$$

Look Pa, Types!

Syntax

The syntax of types is as follows.

(1) $\iota_0, \iota_1, \ldots$ are (basic) types; one for each B_i.

(2) There is a denumerable set of type variables, which are types. We use $\alpha, \beta, \gamma, \ldots$ to range over type variables.

(3) If ρ and σ are types, so is $\rho \to \sigma$.

Basically a Data Type

$$\begin{array}{lcll} \mu, \nu & ::= & \iota_1 \mid \iota_2 \mid \cdots \mid \mu \to \nu & \text{“monotypes”} \\ \rho, \sigma & ::= & \alpha \mid \iota_1 \mid \iota_2 \mid \cdots \mid \rho \to \sigma & \text{“polytypes”} \end{array}$$

$$\text{INT}$$

$$\text{INT} \to \text{BOOL}$$

$$\alpha \to \alpha$$

$$\alpha \to \beta \to \alpha$$

$$\alpha \to \text{INT} \to \alpha$$

Look Pa, Types!

Semantics: when do values have monotypes?

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \bot_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \bot_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Look Pa, Types!

Semantics: when do values have monotypes?

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \perp_V$ or $v \mathbin{\mathsf{E}} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathbin{\mathsf{E}} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Nontermination has any type
(yes, nontermination is a value!!!)

Look Pa, Types!

Semantics: when do values have monotypes?

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \perp_V$ or $v \,\mathsf{E}\, B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \,\mathsf{E}\, F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Basic values have their (obvious) basic type (i.e. true is a Bool, etc.)

Look Pa, Types!

Semantics: when do values have monotypes?

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \perp_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Any function that
returns v's whenever you give it μ's
has type μ -> v

Look Pa, Types!

Semantics: when do values have monotypes?

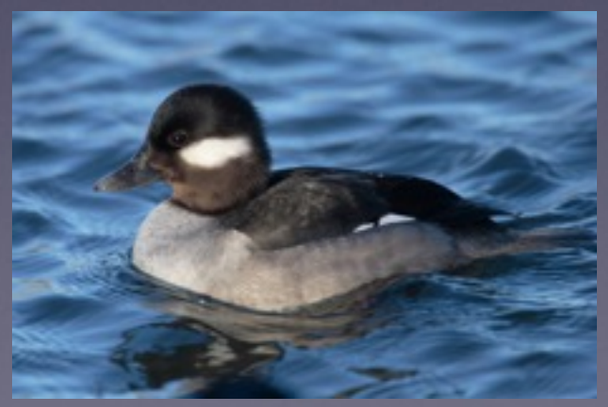
We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota_i$ iff $v = \perp_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Any function that
returns v's whenever you give it μ's
has type μ -> v

Behavioural Definition!



Look Pa, Types!

Semantics: when do values have monotypes?

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

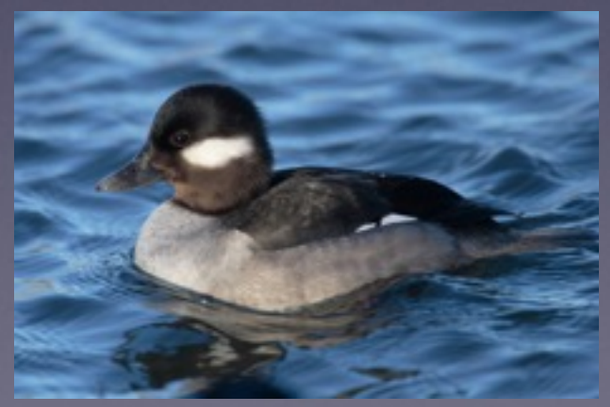
(i) $v : \iota_i$ iff $v = \perp_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Any function that
returns v's whenever you give it μ's
has type μ -> v

Behavioural Definition!

And of course $\perp_V$ has every type



Look Pa, Types!

Semantics: what about polytypes?

Look Pa, Types!

Semantics: what about polytypes?

$$v : \sigma \text{ iff } \forall \mu \leqslant \sigma \cdot v : \mu.$$

Look Pa, Types!

Semantics: what about polytypes?

$$v : \sigma \text{ iff } \forall \mu \leqslant \sigma \cdot v : \mu.$$

In English: a value v has a polytype σ if it has any monotype μ that you get by substituting for σ's type variables.

Look Pa, Types!

Semantics: what about polytypes?

$$v : \sigma \text{ iff } \forall \mu \leqslant \sigma \cdot v : \mu.$$

In English: a value v has a polytype σ if it has any monotype μ that you get by substituting for σ's type variables.

Example:

$$v : \alpha \to \alpha \quad \textit{iff } v : \text{INT} \to \text{INT}$$
$$\textit{and } v : \text{BOOL} \to \text{BOOL}$$
$$\textit{and } v : (\text{INT} \to \text{INT}) \to (\text{INT} \to \text{INT})$$
$$\textit{and } \ldots$$

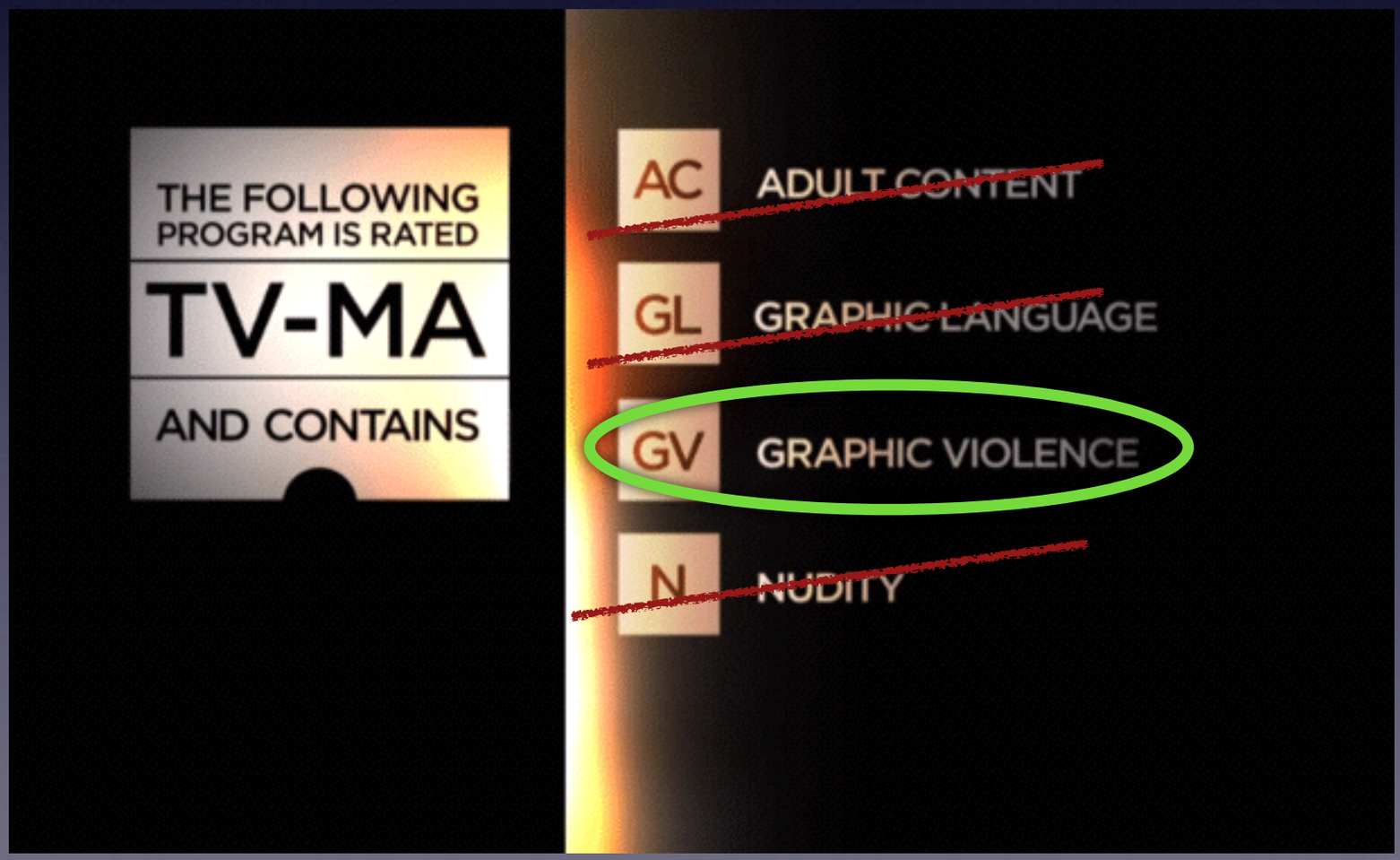
Look Pa, Types!

Semantics: what about types for *expressions?*

Look Pa, Types!

Semantics: what about types for *expressions?*

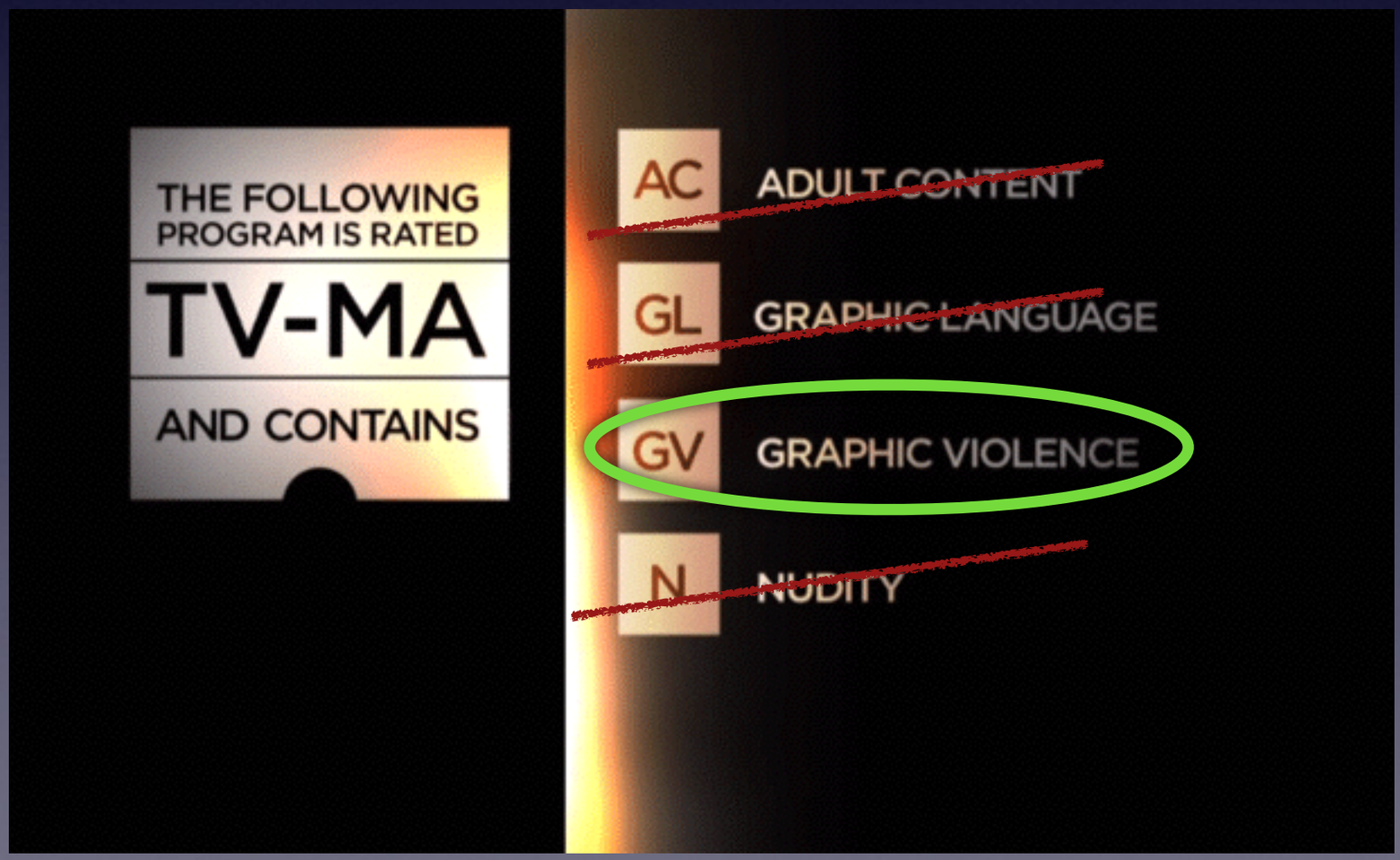
Yes, but…



Look Pa, Types!

Semantics: what about types for *expressions?*

Yes, but…



JOURNAL OF COMPUTER AND SYSTEM SCIENCES 17, 348–375 (1978)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $\mathscr{W}$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

1. INTRODUCTION

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs—anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand a type discipline such as that of ALGOL 68 [22] which precludes the flexibility mentioned above, also precludes the programming style which we are talking about. ALGOL 60 was more flexible—in that it required procedure parameters to be specified only as "*procedure*" (rather than say "*integer to real procedure*") —but the flexibility was not uniform, and not sufficient.

An early discussion of such flexibility can be found in Strachey [19], who was probably the first to call it polymorphism. In fact he qualified it as "parametric" polymorphism, in contrast to what he called "adhoc" polymorphism. An example of the latter is the use of "+" to denote both integer and real addition (in fact it may be further extended to denote complex addition, vector addition, etc.); this use of an identifier at several distinct types is often now called "overloading," and we are not concerned with it in this paper.

In this paper then, we present and justify one method of gaining type flexibility, but also retaining a discipline which ensures robust programs. We have evidence that this

348

0022-0000/78/0173-0348$02.00/0

From List of Bindings...

A *prefix* p is a finite sequence whose members have the form *let* x, *fix* x, or λx,

From List of Bindings…

A *prefix* p is a finite sequence whose members have the form *let* x, *fix* x, or λx,

…To Type Environments!

$\bar{p}$ is an assignment of a type to each element of p, (In modern notation: Γ)

e.g., $\lambda x_{\mathrm{INT}}, \lambda y_{\mathrm{INT}}, let\, x_{\mathrm{BOOL}}$

From List of Bindings…

A *prefix* p is a finite sequence whose members have the form *let* x, *fix* x, or λx,

…To Type Environments!

$\bar{p}$ is an assignment of a type to each element of p, (In modern notation: Γ)

e.g., $\lambda x_{\text{INT}}, \lambda y_{\text{INT}}, let\, x_{\text{BOOL}}$

Type Environments Assign Types to Value Environments!

η *respects* $\bar{p}$ iff, whenever *let* x_ρ or λx_ρ or *fix* x_ρ is active in $\bar{p}$, $\eta[\![x]\!] : \rho$.

Suggestive notation: $\eta : \bar{p}$

From List of Bindings…

A *prefix* p is a finite sequence whose members have the form *let* x, *fix* x, or λx,

…To Type Environments!

$\bar{p}$ is an assignment of a type to each element of p, (In modern notation: Γ)

e.g., $\lambda x_{\text{INT}}, \lambda y_{\text{INT}}, let\, x_{\text{BOOL}}$

Type Environments Assign Types to Value Environments!

η *respects* $\bar{p}$ iff, whenever *let* x_ρ or λx_ρ or *fix* x_ρ is active in $\bar{p}$, $\eta[\![x]\!] : \rho$.

Suggestive notation: $\eta : \bar{p}$

Semantics of Expressions

$\bar{p} \models d : \tau$ iff for all η, *If* η *respects* $\bar{p}$ *then* $\mathscr{E}[\![d]\!]\eta : \tau$.

Semantics of Expressions

$\overline{p} \models d : \tau$ iff for all η, *If* η *respects* $\overline{p}$ *then* $\mathscr{E}[\![d]\!]\eta : \tau$.

Semantics of Expressions

$\bar{p} \models d : \tau$ iff for all η, *If* η *respects* $\bar{p}$ *then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., if every value environment η
that meets the *input spec* p
yields an output value $v = \mathscr{E}[\![d]\!]\eta$
that meets the *output spec* **τ**!

Semantics of Expressions

$\overline{p} \models d : \tau$ iff for all η,

i.e., if every value environment η
that meets the *input spec* p
yields an output value $v = \mathscr{E}[\![d]\!]\eta$
that meets the *output spec* $\boldsymbol{\tau}$!

Static analogue to $v = \mathscr{E}[\![d]\!]\eta$

- Relates a *class* of inputs to a *class* of outputs!

Semantics of Expressions

$\overline{p} \models d : \tau$ iff for all η,

i.e., if every value environment η
that meets the *input spec* p
yields an output value $v = \mathscr{E}[\![d]\!]\eta$
that meets the *output spec* $\boldsymbol{\tau}$!

Static analogue to $v = \mathscr{E}[\![d]\!]\eta$

- Relates a *class* of inputs to a *class* of outputs!

But it's so so so undecidable!

Syntactic Type System

a.k.a. Type Assignment System

a.k.a. Type System

$\bar{p} \mid \bar{e}_\sigma$

(i) $\bar{p} \mid x_\tau$ is wt iff it is standard, and *either*

(a) λx_τ or *fix* x_τ is active in $\bar{p}$, *or*

(b) *let* x_σ is active in $\bar{p}$, and τ is a generic instance of σ.

(ii) $\bar{p} \mid (\bar{e}_\rho \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \mid \bar{e}'$ are both wt, and $\rho = \sigma \to \tau$.

(iii) $\bar{p} \mid (\textit{if } \bar{e}_\rho \textit{ then } \bar{e}'_\sigma \textit{ else } \bar{e}''_\sigma)_{\tau'}$ is wt iff $\bar{p} \mid \bar{e}$, $\bar{p} \mid \bar{e}'$ and $\bar{p} \mid \bar{e}''$ are all wt, $\rho = \iota_0$, and $\sigma = \tau = \tau'$.

(iv) $\bar{p} \mid (\lambda x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \lambda x_\rho \mid \bar{e}$ is wt and $\tau = \rho \to \sigma$.

(v) $\bar{p} \mid (\textit{fix } x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \textit{fix } x_\rho \mid \bar{e}$ is wt and $\rho = \sigma = \tau$.

(vi) $\bar{p} \mid (\textit{let } x_\rho = \bar{e}_\rho \textit{ in } \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \cdot \textit{let } x_\rho \mid \bar{e}'$ are both wt, and $\sigma = \tau$.

Syntactic Type System

a.k.a. Type Assignment System
a.k.a. Type System

$\bar{p} \mid \bar{e}_\sigma$

Modern-ish Notation(*):

$A \vdash e : \sigma$

[Damas & Milner 1982]

(i) $\bar{p} \mid x_\tau$ is wt iff it is standard, and *either*

(a) λx_τ or *fix* x_τ is active in $\bar{p}$, *or*

(b) *let* x_σ is active in $\bar{p}$, and τ is a generic instance of σ.

(ii) $\bar{p} \mid (\bar{e}_\rho \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \mid \bar{e}'$ are both wt, and $\rho = \sigma \to \tau$.

(iii) $\bar{p} \mid (\textit{if } \bar{e}_\rho \textit{ then } \bar{e}'_\sigma \textit{ else } \bar{e}''_\sigma)_{\tau'}$ is wt iff $\bar{p} \mid \bar{e}$, $\bar{p} \mid \bar{e}'$ and $\bar{p} \mid \bar{e}''$ are all wt, $\rho = \iota_0$, and $\sigma = \tau = \tau'$.

(iv) $\bar{p} \mid (\lambda x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \lambda x_\rho \mid \bar{e}$ is wt and $\tau = \rho \to \sigma$.

(v) $\bar{p} \mid (\textit{fix } x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \textit{fix } x_\rho \mid \bar{e}$ is wt and $\rho = \sigma = \tau$.

(vi) $\bar{p} \mid (\textit{let } x_\rho = \bar{e}_\rho \textit{ in } \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \cdot \textit{let } x_\rho \mid \bar{e}'$ are both wt, and $\sigma = \tau$.

$$\text{TAUT: } \frac{}{A \vdash x : \sigma} \ (x : \sigma \in A) \qquad \text{INST: } \frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \ (\sigma > \sigma')$$

$$\text{GEN: } \frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha \sigma} \ (\alpha \text{ not free in } A) \qquad \text{COMB: } \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\, e') : \tau}$$

$$\text{ABS: } \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{LET: } \frac{A \vdash e : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash (\texttt{let}\, x = e\, \texttt{in}\, e') : \tau}$$

Syntactic Type System

a.k.a. Type Assignment System
a.k.a. Type System

$\bar{p} \mid \bar{e}_\sigma$

Modern-ish Notation(*):

$A \vdash e : \sigma$

[Damas & Milner 1982]

(i) $\bar{p} \mid x_\tau$ is wt iff it is standard, and *either*

(a) λx_τ or *fix* x_τ is active in $\bar{p}$, *or*

(b) *let* x_σ is active in $\bar{p}$, and τ is a generic instance of σ.

(ii) $\bar{p} \mid (\bar{e}_\rho \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \mid \bar{e}'$ are both wt, and $\rho = \sigma \to \tau$.

(iii) $\bar{p} \mid (\textit{if } \bar{e}_\rho \textit{ then } \bar{e}'_\sigma \textit{ else } \bar{e}''_\sigma)_{\tau'}$ is wt iff $\bar{p} \mid \bar{e}$, $\bar{p} \mid \bar{e}'$ and $\bar{p} \mid \bar{e}''$ are all wt, $\rho = \iota_0$, and $\sigma = \tau = \tau'$.

(iv) $\bar{p} \mid (\lambda x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \lambda x_\rho \mid \bar{e}$ is wt and $\tau = \rho \to \sigma$.

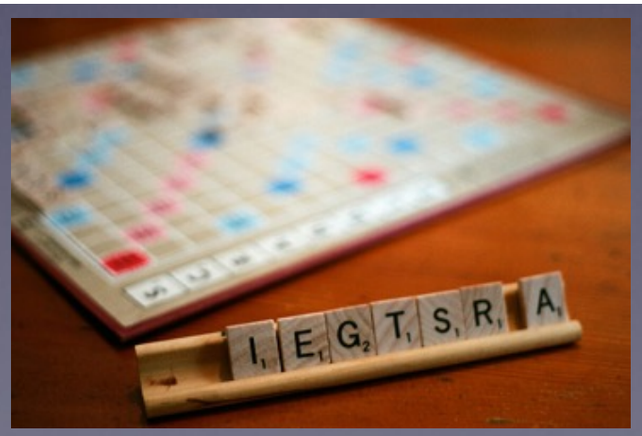
(v) $\bar{p} \mid (\textit{fix } x_\rho \cdot \bar{e}_\sigma)_\tau$ is wt iff $\bar{p} \cdot \textit{fix } x_\rho \mid \bar{e}$ is wt and $\rho = \sigma = \tau$.

(vi) $\bar{p} \mid (\textit{let } x_\rho = \bar{e}_\rho \textit{ in } \bar{e}'_\sigma)_\tau$ is wt iff $\bar{p} \mid \bar{e}$ and $\bar{p} \cdot \textit{let } x_\rho \mid \bar{e}'$ are both wt, and $\sigma = \tau$.

$$\text{TAUT: } \frac{}{A \vdash x : \sigma} \ (x : \sigma \in A) \qquad \text{INST: } \frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \ (\sigma > \sigma')$$

$$\text{GEN: } \frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha \sigma} \ (\alpha \text{ not free in } A) \qquad \text{COMB: } \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\, e') : \tau}$$

$$\text{ABS: } \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{LET: } \frac{A \vdash e : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau}{A \vdash (\texttt{let}\, x = e\, \texttt{in}\, e') : \tau}$$



Basically a Data Type!
(proofs of type assignment)

Comparison

Semantic

We first give the semantics of monotypes; that is, we give the conditions under which a value $v \in V$ *possesses* a monotype μ, which we write $v : \mu$.

(i) $v : \iota$ iff $v = \perp_V$ or $v \mathsf{E} B_i$

(ii) $v : \mu \to \nu$ iff *either* $v = \perp_V$, *or* $v \mathsf{E} F$ and $(v \mid F)u : \nu$ whenever $u : \mu$.

Syntactic (but suggestive!)

$$\text{ABS: } \frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash (\lambda x.e) : \tau' \to \tau} \qquad \text{COMB: } \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\, e') : \tau}$$

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., If $\bar{p} \vdash d : \tau$ then $\bar{p} \models d : \tau$

Data Structure

Behaviour

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

i.e., If $\bar{p} \vdash d : \tau$ then $\bar{p} \models d : \tau$

Data Structure

Behaviour

Every proof of type assignment
says something meaningful about code

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

$$\text{If} \quad \bar{p} \vdash d : \tau \quad \text{then} \quad \bar{p} \models d : \tau$$

Syntax

Semantics

Every proof of type assignment
says something meaningful about code

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

$$\text{If} \quad \bar{p} \vdash d : \tau \quad \text{then} \quad \bar{p} \models d : \tau$$

Syntax

Semantics

As a corollary, under the conditions of the theorem we have

$$\mathscr{E}[\![d]\!]\eta \neq \text{wrong},$$

since wrong has no type.

Big-Picture Payoff!

Semantic Soundness

Connects Type Assignment to Type Semantics

THEOREM 1 (Semantic Soundness). *If* η *respects* $\bar{p}$ *and* $\bar{p} \mid \bar{d}_\tau$ *is well typed then* $\mathscr{E}[\![d]\!]\eta : \tau$.

$$\text{If} \quad \bar{p} \vdash d : \tau \quad \text{then} \quad \bar{p} \models d : \tau$$

Syntax

Semantics

Every proof of type assignment
says something meaningful about code

How can we construct such proofs for our code?

Type Checker

4. A Well-Typing Algorithm and Its Correctness

4.1. *The Algorithm $\mathcal{W}$*

In this section we tackle the question of finding a well typing for a prefixed expression. We present an algorithm $\mathcal{W}$ for this. We would like to prove that $\mathcal{W}$ is both syntactically sound and (in some sense) complete. By syntactic soundness, we mean that whenever $\mathcal{W}$ succeeds it produces a wt; by completeness, we mean that whenever a wt exists, $\mathcal{W}$ succeeds in finding one which is (in some sense) at least as general.

well typing = type assignment (proof)

Type Checker

4. A Well-Typing Algorithm and Its Correctness

4.1. *The Algorithm $\mathscr{W}$*

In this section we tackle the question of finding a well typing for a prefixed expression. We present an algorithm $\mathscr{W}$ for this. We would like to prove that $\mathscr{W}$ is both syntactically sound and (in some sense) complete. By syntactic soundness, we mean that whenever $\mathscr{W}$ succeeds it produces a wt; by completeness, we mean that whenever a wt exists, $\mathscr{W}$ succeeds in finding one which is (in some sense) at least as general.

Algorithm $\mathscr{W}$

$\mathscr{W}(\bar{p}, f) = (T, \bar{f})$, where

(i) If f is x, then:

if λx_σ or *fix* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\sigma$;
if *let* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\tau$
where $\tau = [\beta_i/\alpha_i]\sigma$, α_i are the generic variables of σ, and β_i are new variables.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(iii) If f is (*if* d *then* e *else* e'), then:

let $(R, \bar{d}_o) = \mathscr{W}(\bar{p}, d)$ and $U_0 = \mathscr{U}(\rho, \iota_0)$;
let $(S, \bar{e}_\sigma) = \mathscr{W}(U_0R\bar{p}, e)$, and $(S', \bar{e}'_{\sigma'}) = \mathscr{W}(SU_0R\bar{p}, e')$;
let $U = \mathscr{U}(S'\sigma, \sigma')$;
then $T = US'SU_0R$, and
$\bar{f} = U((\textit{if } S'\bar{e} \textit{ then } S'SU_0\bar{d} \textit{ else } \bar{e}')_\sigma)$.

(iv) If f is $(\lambda x \cdot d)$, then:

let $(R, \bar{d}) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, where β is new;
then $T = R$, and $\bar{f} = (\lambda x_{R\beta} \cdot \bar{d}_\rho)_{R\beta \to \rho}$.

(v) If f is $(\textit{fix } x \cdot d)$, then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \textit{fix } x_\beta, d)$, β new;
let $U = \mathscr{U}(R\beta, \rho)$;
then $T = UR$, and $\bar{f} = (\textit{fix } x_{UR\beta} \cdot U\bar{d})_{UR\beta}$.

(vi) If f is (*let* $x = d$ *in* e), then:

let $(R, \bar{d}_\sigma) = \mathscr{W}(\bar{p}, d)$;
let $(S, e_\sigma) = \mathscr{W}(R\bar{p} \cdot \textit{let } x_\sigma, e)$;
then $T = SR$, and $\bar{f} = (\textit{let } x_{S\sigma} = S\bar{d} \text{ in } \bar{e})_\sigma$. ∎

Some Computational Procedure or Other
(see paper for details)

Type Checker

4. A Well-Typing Algorithm and Its Correctness

4.1. *The Algorithm $\mathscr{W}$*

In this section we tackle the question of finding a well typing for a prefixed expression. We present an algorithm $\mathscr{W}$ for this. We would like to prove that $\mathscr{W}$ is both syntactically sound and (in some sense) complete. By syntactic soundness, we mean that whenever $\mathscr{W}$ succeeds it produces a wt; by completeness, we mean that whenever a wt exists, $\mathscr{W}$ succeeds in finding one which is (in some sense) at least as general.

Algorithm $\mathscr{W}$

$\mathscr{W}(\bar{p}, f) = (T, \bar{f})$, where

(i) If f is x, then:

if λx_σ or *fix* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\sigma$;
if *let* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\tau$
where $\tau = [\beta_i/\alpha_i]\sigma$, α_i are the generic variables of σ, and β_i are new variables.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(iii) If f is (*if* d *then* e *else* e'), then:

let $(R, \bar{d}_o) = \mathscr{W}(\bar{p}, d)$ and $U_0 = \mathscr{U}(\rho, \iota_0)$;
let $(S, \bar{e}_\sigma) = \mathscr{W}(U_0R\bar{p}, e)$, and $(S', \bar{e}'_{\sigma'}) = \mathscr{W}(SU_0R\bar{p}, e')$;
let $U = \mathscr{U}(S'\sigma, \sigma')$;
then $T = US'SU_0R$, and
$\bar{f} = U((\textit{if } S'SU_0\bar{d} \textit{ then } S'\bar{e} \textit{ else } \bar{e}')_\sigma)$.

(iv) If f is $(\lambda x \cdot d)$, then:

let $(R, \bar{d}) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, where β is new;
then $T = R$, and $\bar{f} = (\lambda x_{R\beta} \cdot \bar{d}_\rho)_{R\beta \to \rho}$.

(v) If f is $(\textit{fix } x \cdot d)$, then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \textit{fix } x_\beta, d)$, β new;
let $U = \mathscr{U}(R\beta, \rho)$;
then $T = UR$, and $\bar{f} = (\textit{fix } x_{UR\beta} \cdot U\bar{d})_{UR\beta}$.

(vi) If f is (*let* $x = d$ *in* e), then:

let $(R, \bar{d}_\sigma) = \mathscr{W}(\bar{p}, d)$;
let $(S, e_\sigma) = \mathscr{W}(R\bar{p} \cdot \textit{let } x_\rho, e)$;
then $T = SR$, and $\bar{f} = (\textit{let } x_{S\rho} = S\bar{d} \text{ in } \bar{e})_\sigma$. ∎

Some Computational Procedure or Other
(see paper for details)

Is it any good?

Syntactic Soundness

YES!

THEOREM 2 (Syntactic Soundness). *Let $\bar{p}$ be a standard prefix, and $p \mid f$ a (closed) pe. Then, if $\mathscr{W}(\bar{p}, f) = (T, \tilde{f}_\tau)$,*

(A) $T\bar{p} \mid \tilde{f}$ *is wt,*

(B) $\mathrm{Inv}(T) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

and

(C) $\mathrm{Vars}(\tau) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

where New is the set of new type variables used by $\mathscr{W}$.

Connects the results of
Type Checking to the definition of Type Assignment

Syntactic Soundness

THEOREM 2 (Syntactic Soundness). *Let $\bar{p}$ be a standard prefix, and $p \mid f$ a (closed) pe. Then, if $\mathscr{W}(\bar{p}, f) = (T, \tilde{f}_\tau)$,*

(A) $T\bar{p} \mid \tilde{f}$ *is wt,*

(B) $\mathrm{Inv}(T) \subseteq \mathrm{Spec}(\bar{p}) \cup \mathit{New}$,

and

(C) $\mathrm{Vars}(\tau) \subseteq \mathrm{Spec}(\bar{p}) \cup \mathit{New}$,

where New is the set of new type variables used by $\mathscr{W}$.

If *Algorithm W* returns a type assignment,

Syntactic Soundness

THEOREM 2 (Syntactic Soundness). *Let $\bar{p}$ be a standard prefix, and $p \mid f$ a (closed) pe. Then, if $\mathscr{W}(\bar{p}, f) = (T, \tilde{f}_\tau)$,*

(A) $T\bar{p} \mid \tilde{f}$ *is wt,*

(B) $\mathrm{Inv}(T) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

and

(C) $\mathrm{Vars}(\tau) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

where New is the set of new type variables used by $\mathscr{W}$.

If Algorithm W returns a type assignment, then / can build the corresponding proof

Syntactic Soundness

THEOREM 2 (Syntactic Soundness). *Let $\bar{p}$ be a standard prefix, and $p \mid f$ a (closed) pe. Then, if $\mathscr{W}(\bar{p}, f) = (T, \tilde{f}_\tau)$,*

(A) $T\bar{p} \mid \tilde{f}$ *is wt,*

(B) $\mathrm{Inv}(T) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

and

(C) $\mathrm{Vars}(\tau) \subseteq \mathrm{Spec}(\bar{p}) \cup New,$

where New is the set of new type variables used by $\mathscr{W}$.

If Algorithm W returns a type assignment,
then / can build the corresponding proof
(the algorithm essentially does but discards it)

Synopsis

Computation — Type Checker

Syntactic Soundness

Syntax — Type Assignment

Semantic Soundness

Semantics — Semantics of Types

“Well-typed Programs Don’t Go Wrong”

Synopsis

Computation — Type Checker *Often "Obvious"*

Syntactic Soundness

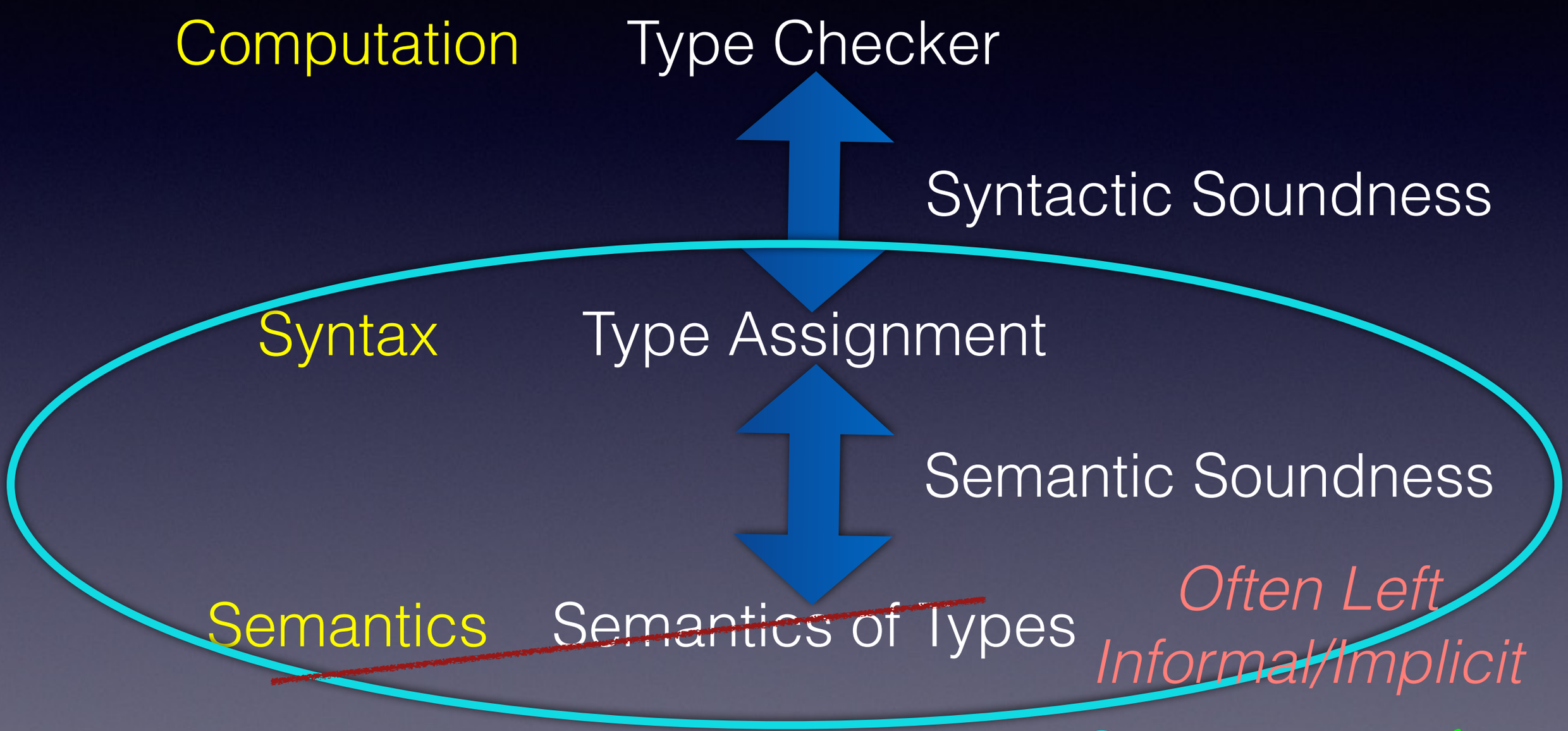
Syntax — Type Assignment

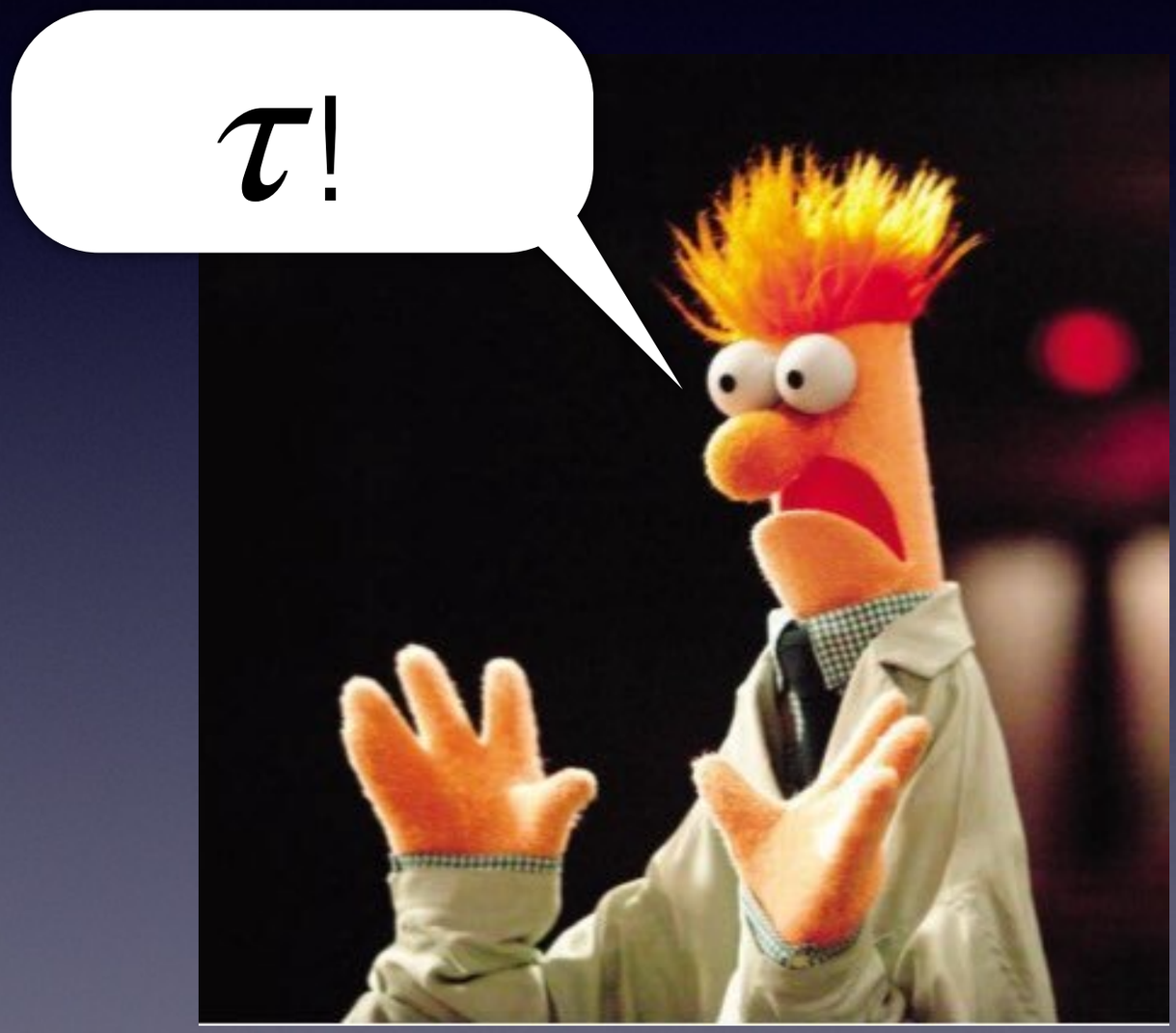
Semantic Soundness

Semantics — Semantics of Types

"Well-typed Programs Don't Go Wrong"

Synopsis


Computation
Type Checker
Syntactic Soundness
Syntax
Type Assignment
Semantic Soundness
Semantics
Semantics of Types
Often Left
Informal/Implicit
"Well-typed Programs Don't Go Wrong"


τ!

What are Types?

- Programming Languages have:
- Syntax
- Semantics
- Pragmatics

What are Types?

- *Types* have:
- Syntax
- Semantics
- Pragmatics

What are Types?

- *Types* have:
  - Syntax
  - Semantics
  - Pragmatics

A type system *is a language…*

What are Types?

- *Types* have:
  - Syntax
  - Semantics
  - Pragmatics

A type system *is a language* for "talking about" properties of programs and program fragments!

A Whirlwind Tour

A Chronology (4 decades)

References:

Types Are Not Sets
by James H. Morris (1973)

A Theory of Type Polymorphism in Programming
by Robin Milner (1978)

Types, Abstraction, and Parametric Polymorphism
by John Reynolds (1983)

Co-induction in Relational Semantics
by Robin Milner and Mads Tofte (1990)

Typing First-class Continuations in ML
by Bruce Duba, Robert Harper, and David MacQueen (1991)

A Syntactic Approach to Type Soundness
by Andrew Wright and Matthias Felleisen (1994)

Semantics of Types for Mutable State
by Amal Ahmed (2004)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

Turing Award (1991)

Did not have a PhD!

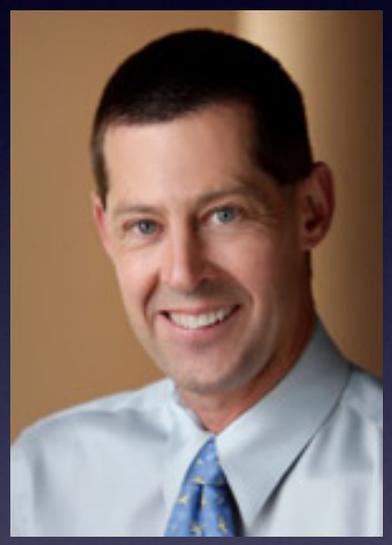
"The Hindley/Milner Type Inference Paper"

A Conceptual Blueprint for Analyzing Types!

A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

Department of Computer Science, Rice University,
Houston, Texas 77251-1892



A *Practical* Blueprint for Analyzing Types!

"Well-typed Programs Don't Go Wrong" ✔

A Chronology (4 decades)

References:

Types Are Not Sets
by James H. Morris (1973)

A Theory of Type Polymorphism in Programming
by Robin Milner (1978)

Types, Abstraction, and Parametric Polymorphism
by John Reynolds (1983)

Co-induction in Relational Semantics
by Robin Milner and Mads Tofte (1990)

Typing First-class Continuations in ML
by Bruce Duba, Robert Harper, and David MacQueen (1991)

A Syntactic Approach to Type Soundness
by Andrew Wright and Matthias Felleisen (1994)

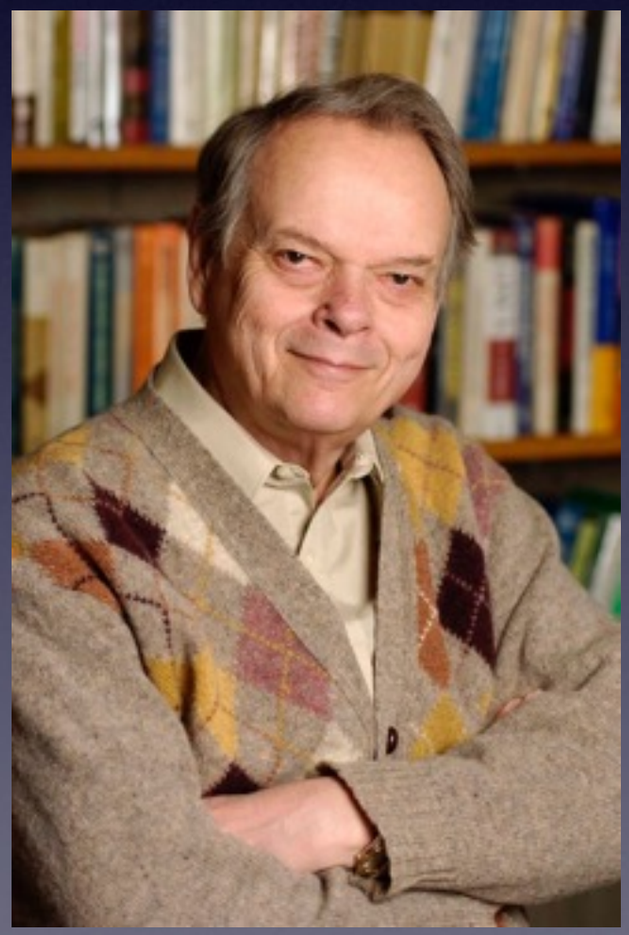
Semantics of Types for Mutable State
by Amal Ahmed (2004)

TYPES ARE NOT SETS*

James H. Morris, Jr.
Xerox Corporation
Palo Alto Research Center (PARC)
Palo Alto, California 94304

TYPES, ABSTRACTION AND PARAMETRIC POLYMORPHISM†

John C. REYNOLDS
Syracuse University
Syracuse, New York, USA



Types can describe and enforce strong behavioural properties (e.g. information-hiding)

A Chronology (4 decades)

References:

Types Are Not Sets
by James H. Morris (1973)

A Theory of Type Polymorphism in Programming
by Robin Milner (1978)

Types, Abstraction, and Parametric Polymorphism
by John Reynolds (1983)

Co-induction in Relational Semantics
by Robin Milner and Mads Tofte (1990)

Typing First-class Continuations in ML
by Bruce Duba, Robert Harper, and David MacQueen (1991)

A Syntactic Approach to Type Soundness
by Andrew Wright and Matthias Felleisen (1994)

Semantics of Types for Mutable State
by Amal Ahmed (2004)

Semantics of Types for Mutable State

Amal Jamil Ahmed

Practical Blueprint for a Semantic Approach to Analyzing Types

What are Types?

- *Types* have:
  - Syntax
  - Semantics
  - Pragmatics

A type system *is a language* for "talking about" properties of programs and program fragments!

Photo Attribution

Dave and Margie Hill: Basai-dai
Hans Olofson: The virgin chastising the Christ child (Max Ernst)
Lerner et al. Searching For Type-Error Messages
Matt Thorpe: Reading