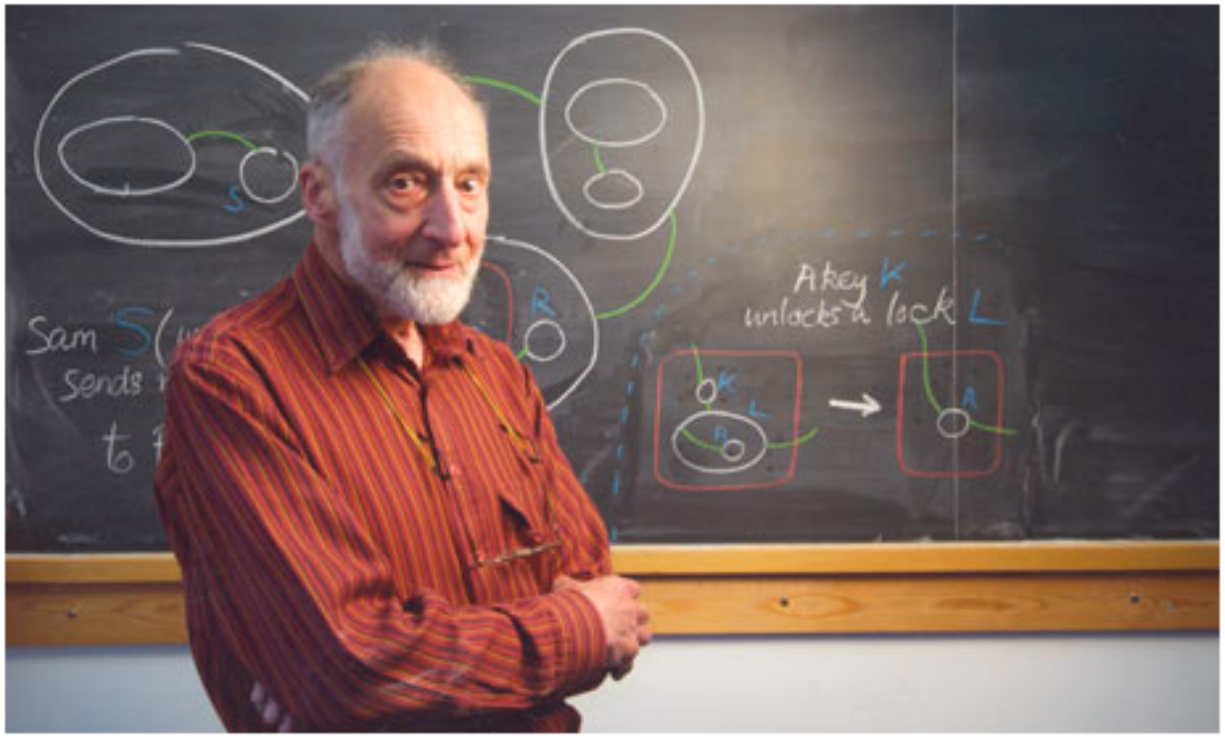
"Types are the leaven of computer programming: they make it digestible."
- R. Milner

Types *á la* Milner

Benjamin C. Pierce
University of Pennsylvania

May 2012

Robin Milner (1934-2010)



A.M. TURING AWARD

For three distinct and complete achievements:

1. **LCF**, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
2. **ML**, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
3. **CCS**, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

A.M. TURING AWARD

For three distinct and complete achievements:

1. **LCF**, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
2. **ML**, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
3. **CCS**, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

A.M. TURING AWARD

For three distinct and complete achievements:

1. **LCF**, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;
2. **ML**, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;
3. **CCS**, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

A.M. TURING AWARD

For three distinct and complete achievements:

1. **LCF**, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. **ML**, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. **CCS**, a general theory of concurrency $\longrightarrow$ **Pi-Calculus**

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

Milner and me

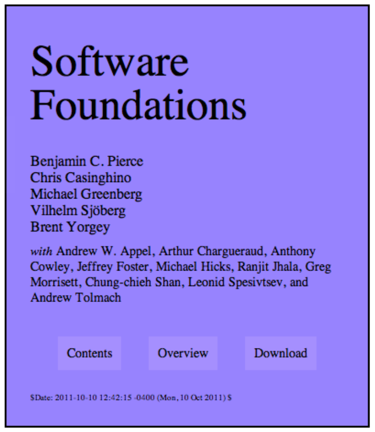
- Last ML postdoc at Edinburgh
  - and one of the first Cambridge postdocs, with Peter Sewell
- Satisfied ML user

Unison

- Pi-calculus type systems (with Davide Sangiorgi)
- Pict programming language (with David Turner)

$$\frac{\text{lambda-calculus}}{\text{ML, Haskell, Scheme, ...}} = \frac{\text{pi-calculus}}{\text{Pict}}$$

- Local type inference → Scala
- POPLMark and Software Foundations



Software Foundations

Benjamin C. Pierce
Chris Casinghino
Michael Greenberg
Vilhelm Sjöberg
Brent Yorgey

with Andrew W. Appel, Arthur Chargueraud, Anthony Cowley, Jeffrey Foster, Michael Hicks, Ranjit Jhala, Greg Morrisett, Chung-chieh Shan, Leonid Spesivtsev, and Andrew Tolmach

$Date: 2011-10-10 12:42:15 -0400 (Mon, 10 Oct 2011) $

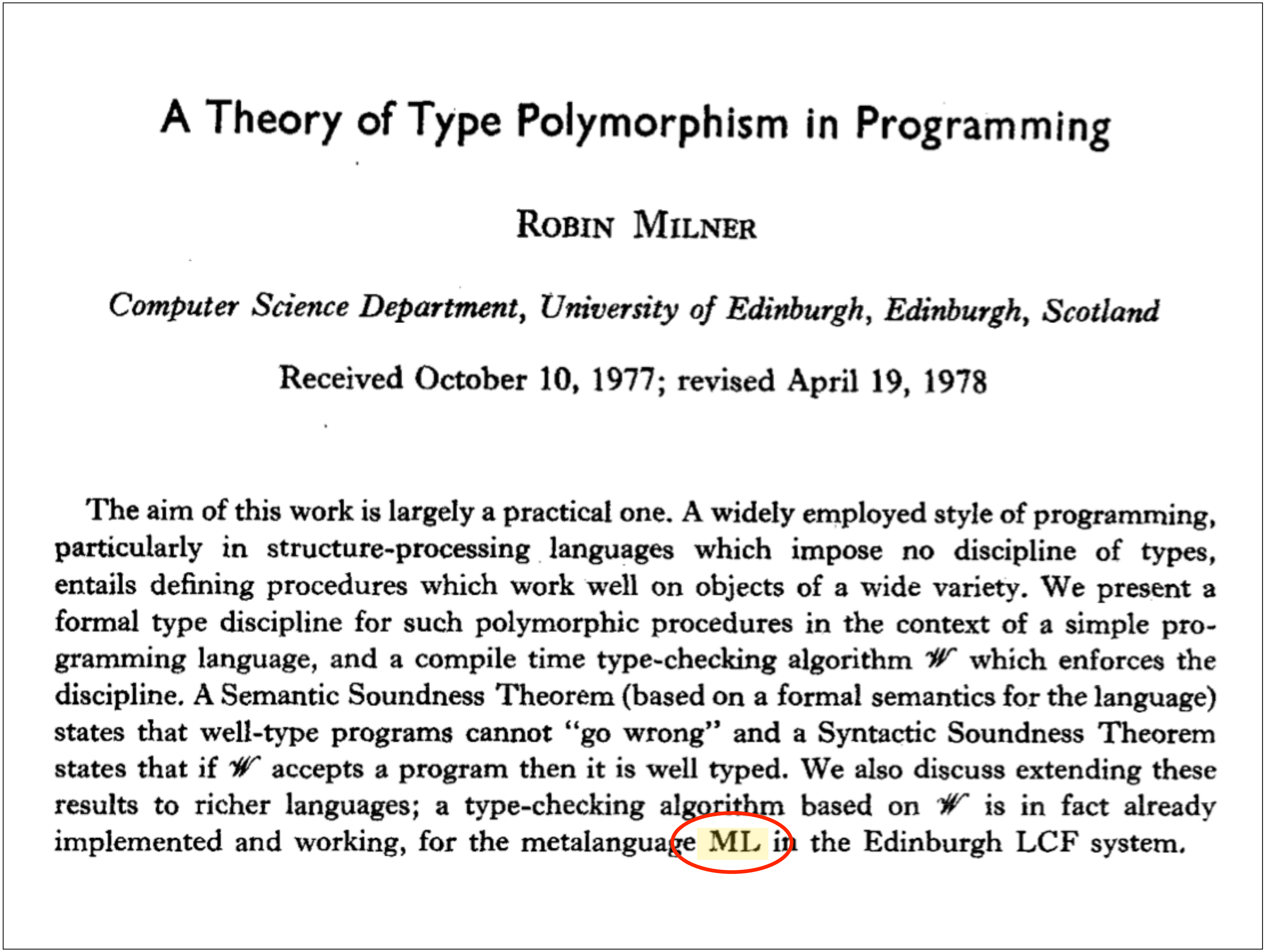
Type inference

Abstract types

Types *á la* Milner

Types for interaction

Types for privacy



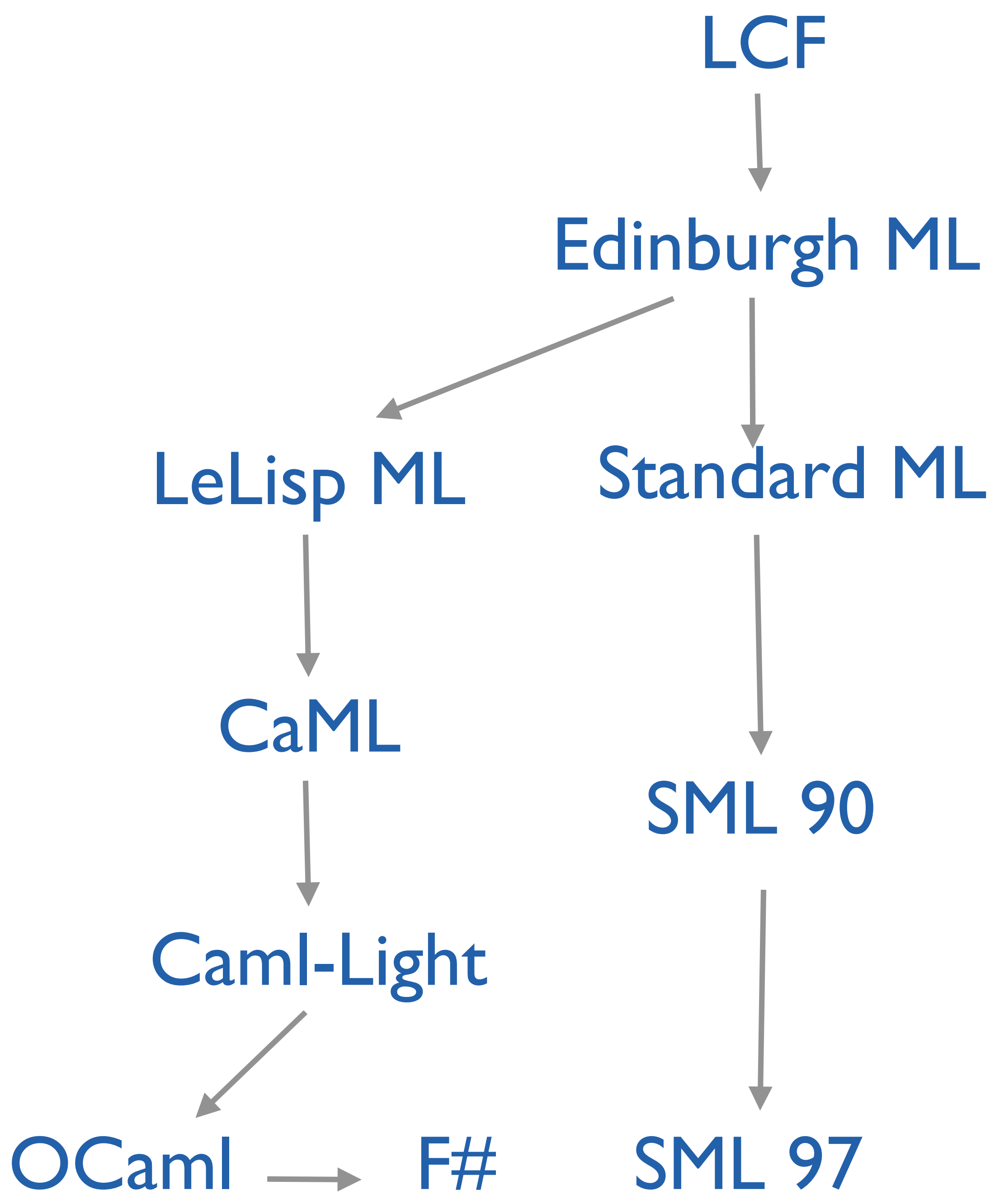
A Theory of Type Polymorphism in Programming

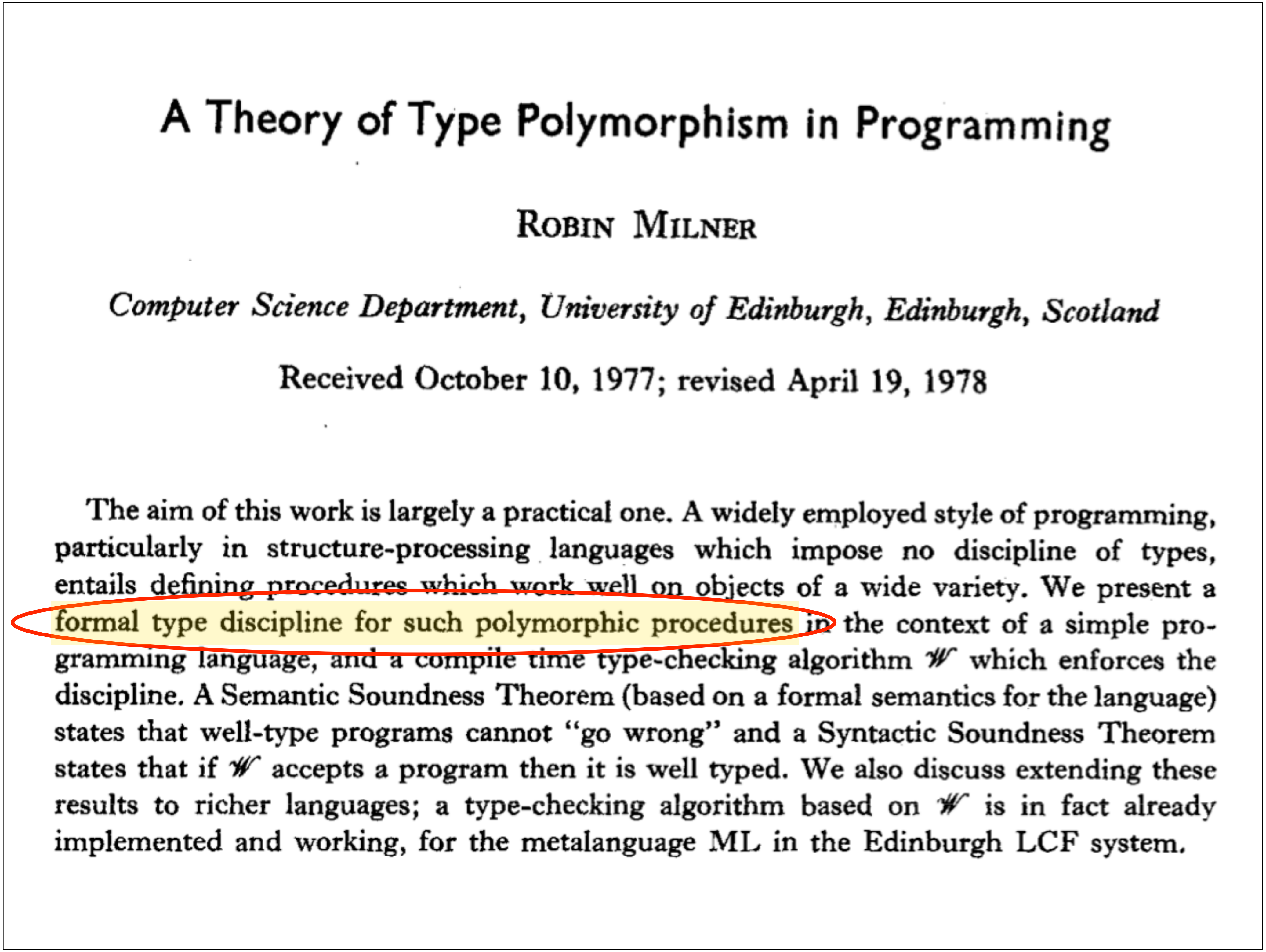
ROBIN MILNER

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.


LCF
Edinburgh ML
LeLisp ML
Standard ML
CaML
SML 90
Caml-Light
OCaml
F#
SML 97



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.

Polymorphism

Consider the *list mapping* function

$$\begin{aligned} \textit{letrec}\ \mathrm{map}(f, m) = {} & \textit{if}\ \mathrm{null}\,(m)\ \textit{then}\ \mathrm{nil} \\ & \textit{else}\ \mathrm{cons}\,(f\,(hd(m)),\ \mathrm{map}\,(f,\ tl(m))) \end{aligned}$$

For example:

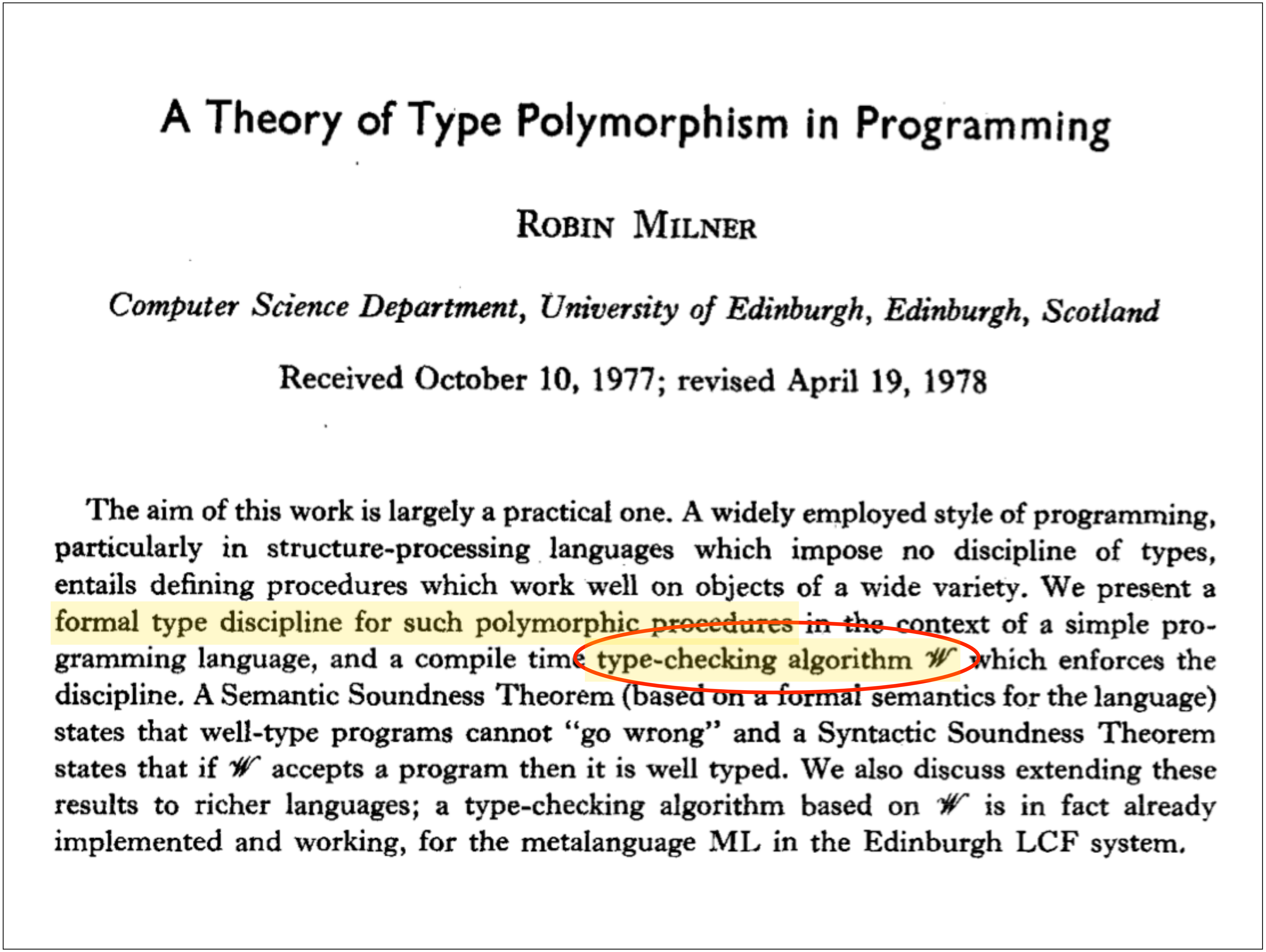
map(square, [1,2,3]) = [1,4,9]

A good type for map is:

$$((\alpha \rightarrow \beta) \times \alpha\ \textit{list}) \rightarrow \beta\ \textit{list}$$

Roots of Polymorphism

- Theory: Girard, Reynolds, Plotkin,...
- Practice: Morris, Liskov, Burstall, MacQueen, ...
- Languages: CLU, Euclid, Simula, Alphard, Hope, ...



A Theory of Type Polymorphism in Programming

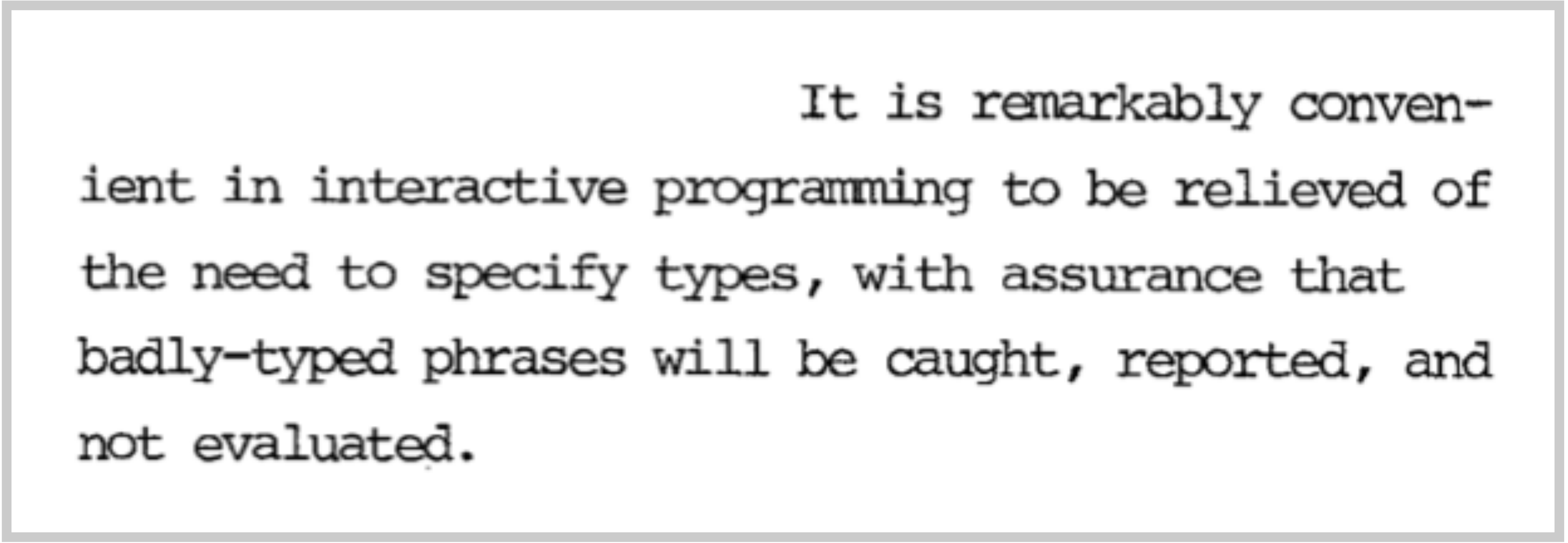
ROBIN MILNER

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.

Type inference


It is remarkably convenient in interactive programming to be relieved of the need to specify types, with assurance that badly-typed phrases will be caught, reported, and not evaluated.

A Metalanguage for interactive proof in LCF
M. Gordon, R. Milner, L. Morris, M. Newey, C. Wadsworth
(POPL 1982)

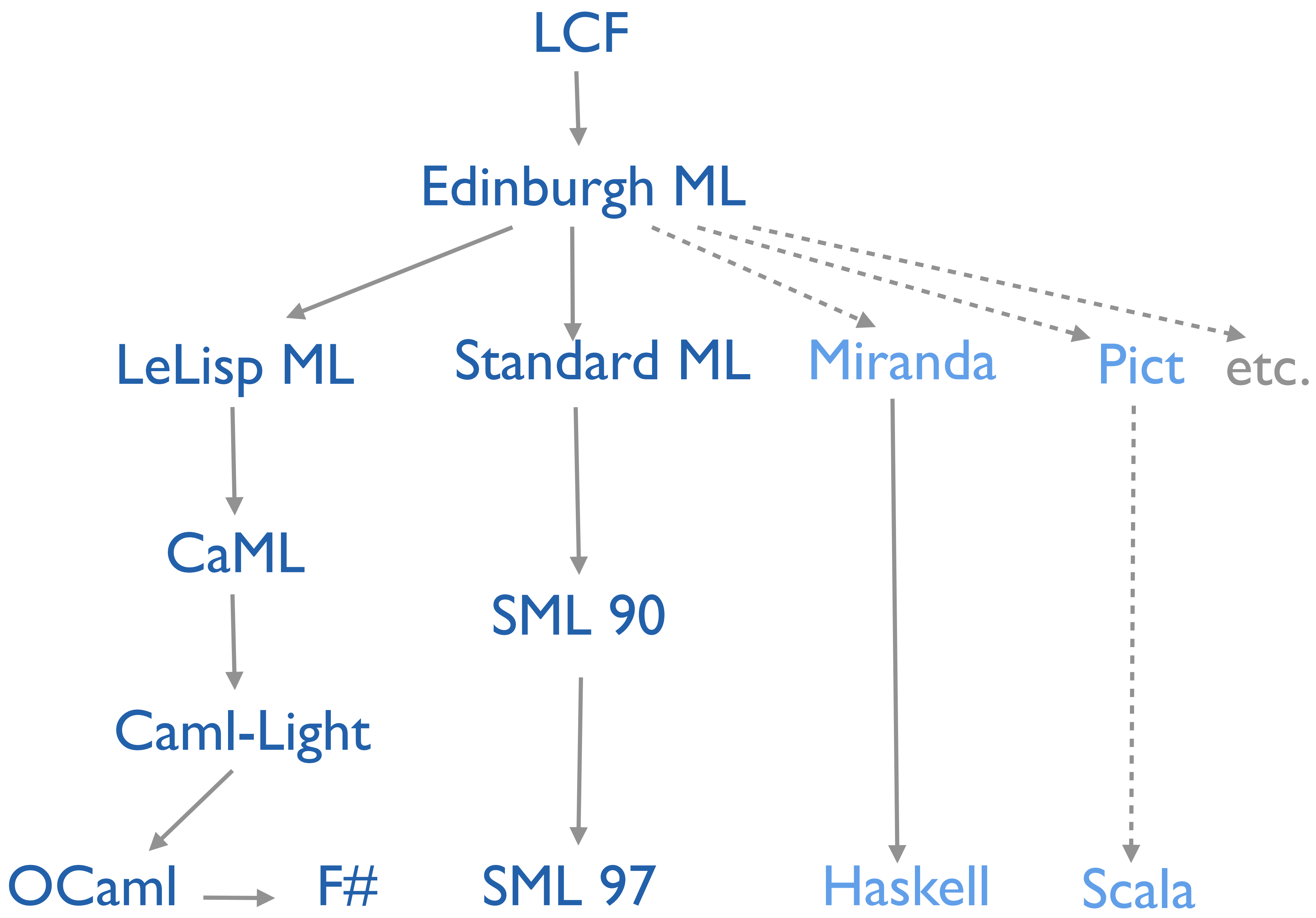
$$\textit{letrec}\ \mathrm{map}(f, m) = \textit{if}\ \mathrm{null}\,(m)\ \textit{then}\ \mathrm{nil}$$
$$\textit{else}\ \mathrm{cons}\,(f\,(hd(m)),\ \mathrm{map}\,(f,\ tl(m)))$$

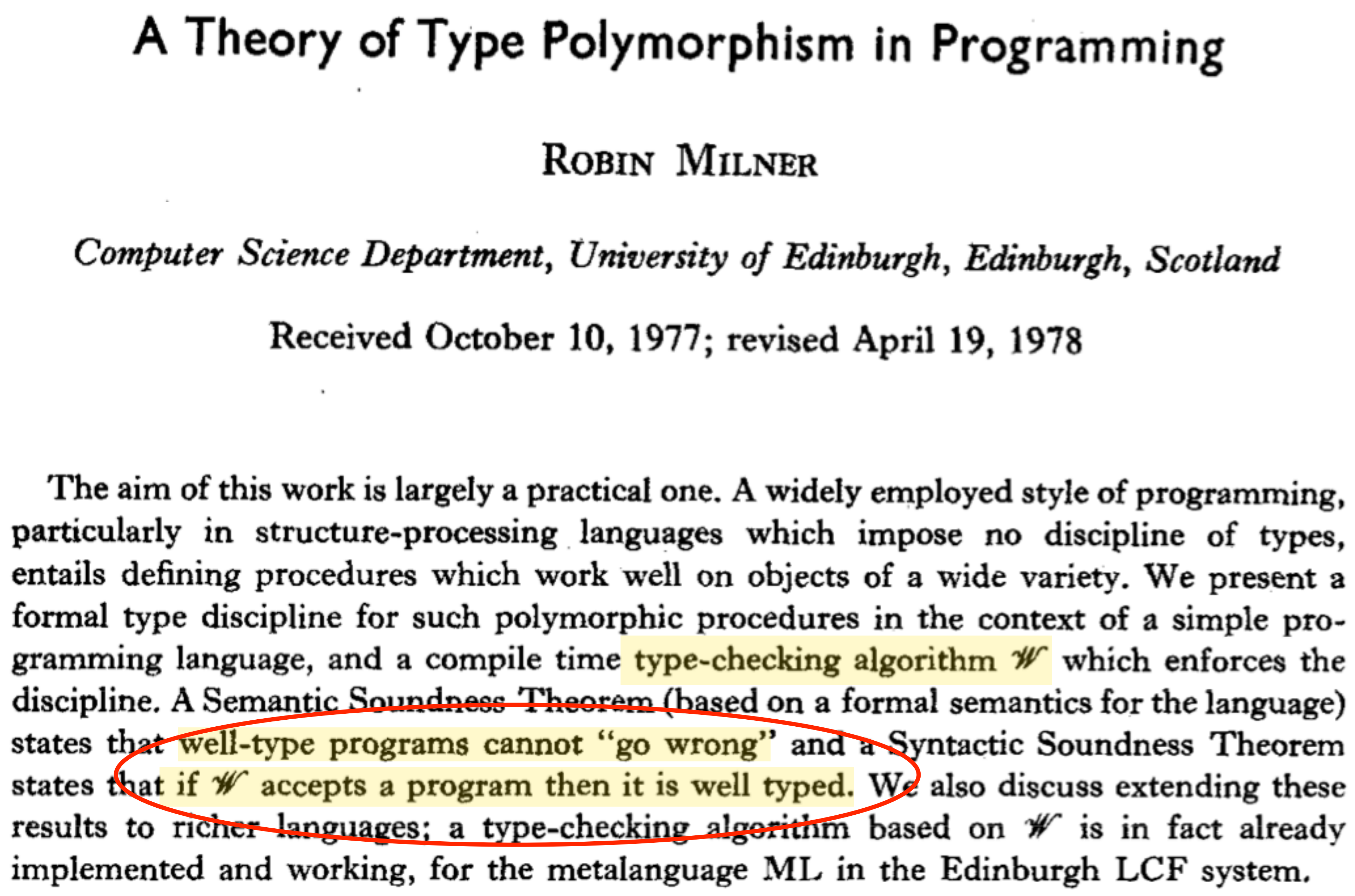
$$\begin{aligned}
\sigma_{\mathrm{null}} &= \tau_1\ \textit{list} \to \textit{bool},\\
\sigma_{\mathrm{nil}} &= \tau_2\ \textit{list},\\
\sigma_{\mathrm{hd}} &= \tau_3\ \textit{list} \to \tau_3\ ,\\
\sigma_{\mathrm{tl}} &= \tau_4\ \textit{list} \to \tau_4\ \textit{list},\\
\sigma_{\mathrm{cons}} &= (\tau_5 \times \tau_5\ \textit{list}) \to \tau_5\ \textit{list}
\end{aligned}$$

$$\begin{aligned}
\sigma_{\mathrm{map}} &= \sigma_{\mathrm{f}} \times \sigma_{\mathrm{m}} \to \rho_1\ ,\\
\sigma_{\mathrm{null}} &= \sigma_{\mathrm{m}} \to \textit{bool},\\
\sigma_{\mathrm{hd}} &= \sigma_{\mathrm{m}} \to \rho_2\ ,\\
\sigma_{\mathrm{tl}} &= \sigma_{\mathrm{m}} \to \rho_3\ ,\\
\sigma_{\mathrm{f}} &= \rho_2 \to \rho_4\ ,\\
\sigma_{\mathrm{map}} &= \sigma_{\mathrm{f}} \times \rho_3 \to \rho_5\ ,\\
\sigma_{\mathrm{cons}} &= \rho_4 \times \rho_5 \to \rho_6\ ,\\
\rho_1 &= \sigma_{\mathrm{nil}} = \rho_6\ .
\end{aligned}$$

Most general solution:

$$\sigma_{\mathrm{map}} = (\gamma \to \delta) \times \gamma\ \textit{list} \to \delta\ \textit{list}$$


LCF
Edinburgh ML
LeLisp ML
Standard ML
Miranda
Pict
etc.
CaML
SML 90
Caml-Light
OCaml
F#
SML 97
Haskell
Scala



A Theory of Type Polymorphism in Programming

ROBIN MILNER

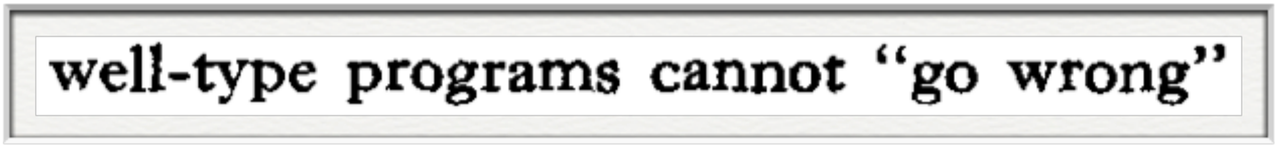
Computer Science Department, University of Edinburgh, Edinburgh, Scotland

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.

Type Soundess

1. Give a denotational semantics assigning each expression e a *meaning* [[e]] in some mathematical domain of abstract computations
   - ... including a special element *wrong* for expressions that fail when evaluated
2. Show that if the type inference algorithm assigns a type τ to some expression e, then e has type τ
3. Associate each type τ with a set of expression meanings [[τ]]
4. Show that if expression e has type τ, then [[e]] $\in$ [[τ]]


well-type programs cannot "go wrong"

Types and behavior

- some types describe *structure:*

  {name: String, age: Int, email: String}

- others constrain the *behavior* of programs...

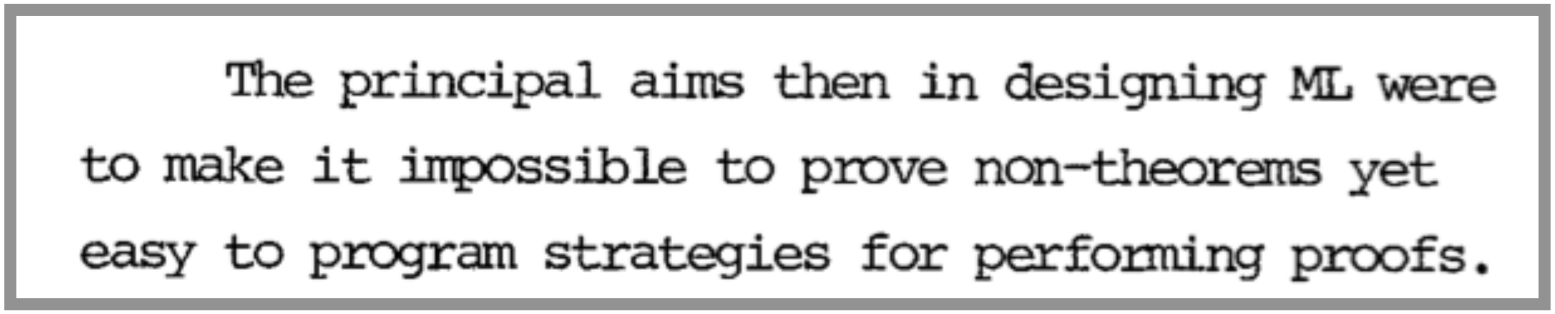
  $$((\alpha \rightarrow \beta) \times \alpha\ \mathit{list}) \rightarrow \beta\ \mathit{list}$$

- ...and their environments:

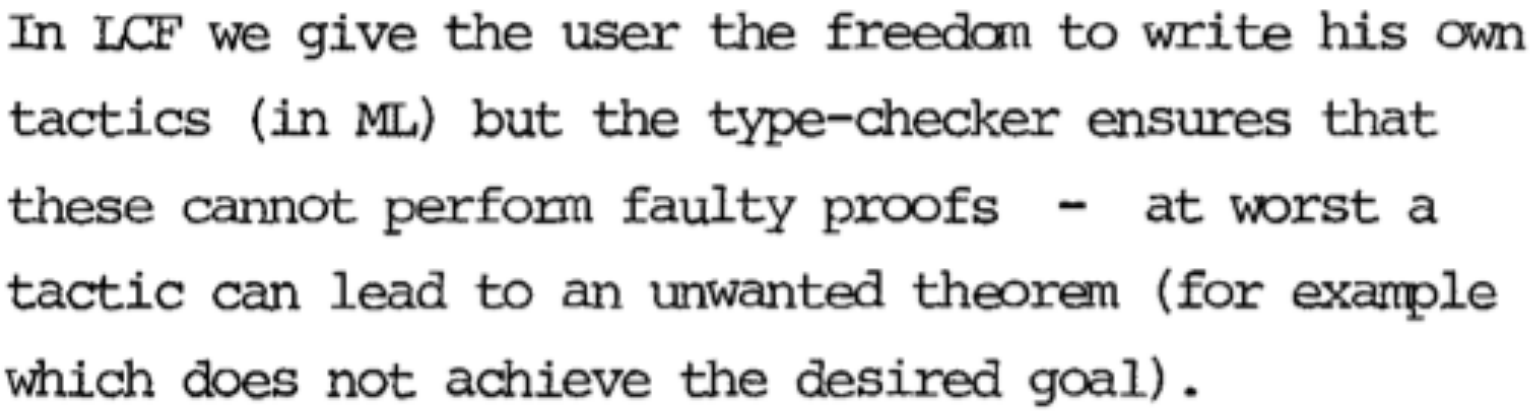
  $$((\alpha \rightarrow \beta) \times \alpha\ \mathit{list}) \rightarrow \beta\ \mathit{list}$$

type ≈ contract between program and environment

Abstract types are behavioral invariants



The principal aims then in designing ML were to make it impossible to prove non-theorems yet easy to program strategies for performing proofs.



In LCF we give the user the freedom to write his own tactics (in ML) but the type-checker ensures that these cannot perform faulty proofs - at worst a tactic can lead to an unwanted theorem (for example which does not achieve the desired goal).

An abstract type of theorems

LCF is basically a programming language (ML) with a predefined abstract type of <u>theorems</u>

```
abstype thm with
  ASSUME : formula → thm
  GEN    : thm → thm
  TRANS  : thm → thm → thm
  ...
```

`ASSUME f`
constructs a proof of
$f \vdash f$

`GEN x w`
constructs a proof of
$\Gamma \vdash \forall x.f$
from a proof of $\Gamma \vdash f$
provided `x` is not free in Γ

`TRANS w1 w2`
constructs a proof of
$\Gamma \vdash t1=t3$
from a proof `w1` of $\Gamma \vdash t1=t2$
and a proof `w2` of $\Gamma \vdash t2=t3$

An abstract type of theorems

LCF is basically a programming language (ML) with a predefined abstract type of <u>theorems</u>

```
abstype thm with
  ASSUME : formula → thm
  GEN    : thm → thm
  TRANS  : thm → thm → thm
  ...
```

Code outside of the `abstype`'s implementation can *only* build theorems by calling these functions!

Types for Interaction

lambda-calculus [Church, 1940s]	pi-calculus [Milner, Parrow, Walker, 1989]
core calculus of functional computation	core calculus of concurrent processes, communicating with messages over channels
everything is a function • all arguments and results of functions are functions	everything is processes and channels • the only thing processes do is communicate over channels • the data exchanged when processes communicate is just a tuple of channels
all computation is function application	all computation is communication
common data and control structures encodable	common data and control structures encodable... including functions!

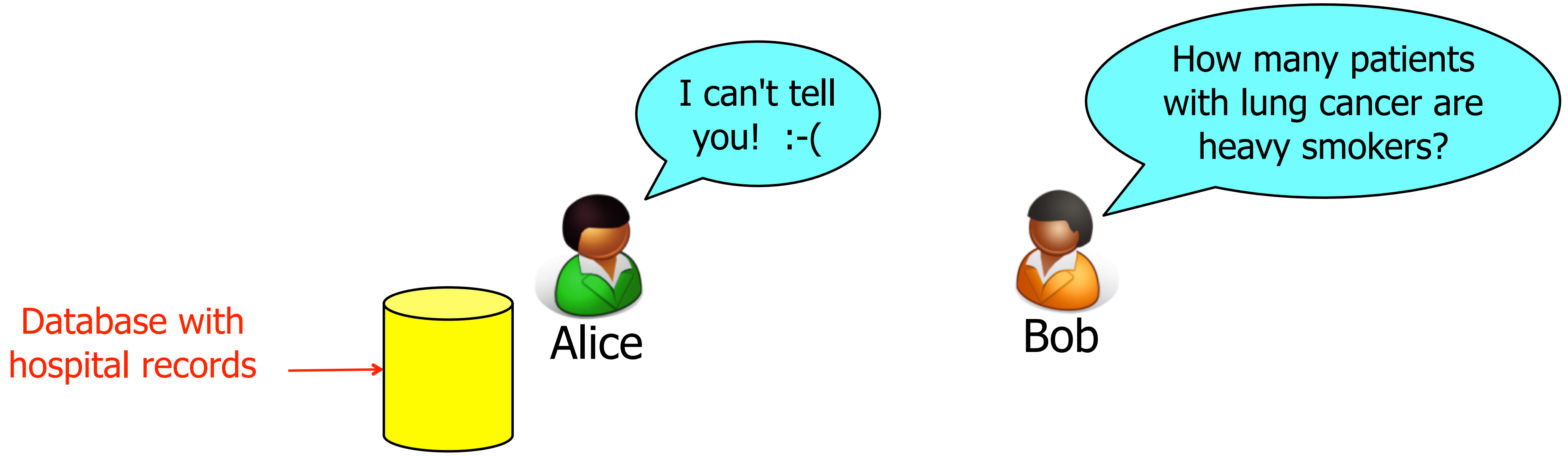
Behavioral types

- [1991] Milner's simple "sort discipline" for the pi-calculus
- [1993ff] "Transplanted" type systems from lambda-calculus
  - subtyping
  - polymorphic / abstract types
  - linear types ("one-shot" channels)
- [2000s] Session types, choreography types

Types for Privacy

Joint work with Jason Reed, Andreas Haeberlen, Marco Gaboardi, Arjun Narayan, ...

Motivation: querying private data

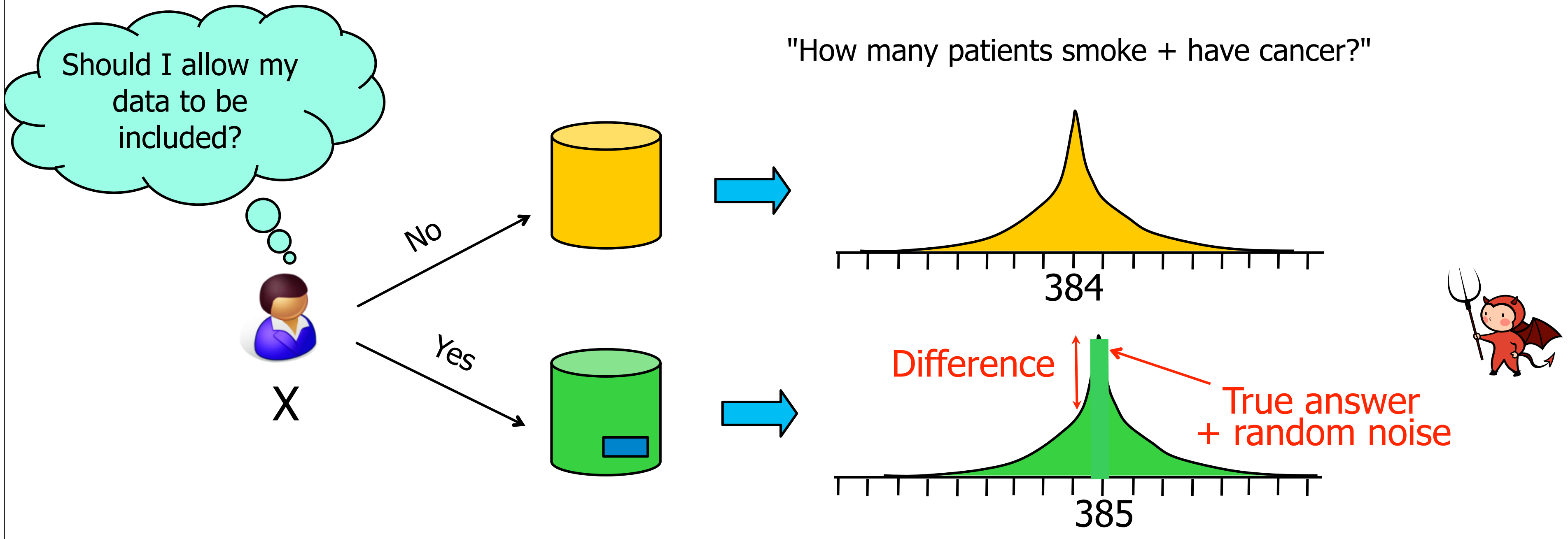

Database with hospital records
I can't tell you! :-(
Alice
How many patients with lung cancer are heavy smokers?
Bob

- A vast trove of data is accumulating in databases
- This data could be useful for many things
  - Example: Use hospital records for medical studies
- But how to release it without violating privacy?

Privacy is hard!

- Idea #1: Anonymize the data
  - "Patient #147, DOB 11/08/1965, zip code 19104, smokes and has lung cancer"
  - What fraction of the U.S. population is uniquely identified by their ZIP code and their full DOB? **63.3%**
  - Another example: Netflix dataset de-anonymized in 2008
- Idea #2: Aggregate the data
  - "385 patients both smoke and have lung cancer"
  - Problem: Someone might know that 384 patients smoke + have cancer, but isn't sure about Benjamin
- Need a more principled approach!

Approach: Differential privacy

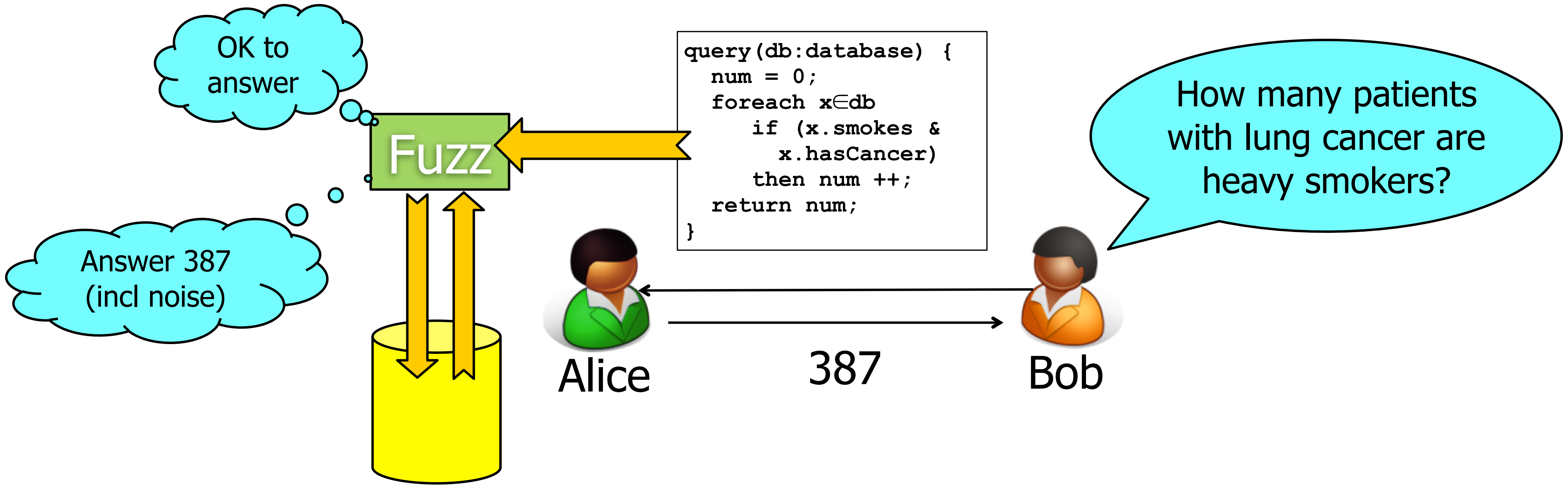

Should I allow my data to be included?
X
No
Yes
"How many patients smoke + have cancer?"
384
Difference
True answer + random noise
385

- Idea: Add a bit of noise to the answer
  - "387 patients smoke + have cancer, plus or minus 3"
- Can bound how much information is leaked
  - Even under worst-case assumptions!

Problem: How much noise?

- What if someone asks the following:
  - "What is the number of people in the database who are called Andreas, multiplied by 1,000,000"
- How do we know...
  - whether it is okay to answer this (given our bound)?
  - and, if so, how much noise we need to add?
- Analysis can be done manually...
  - Example: McSherry/Mironov [KDD'09] on Netflix data
- ... but this does not scale!
  - Each database owner would have to hire a 'privacy expert'
  - Analysis is nontrivial - what if the expert makes a mistake?

The Fuzz system



```
query(db:database) {
  num = 0;
  foreach x∈db
     if (x.smokes &
        x.hasCancer)
     then num ++;
  return num;
}
```

- We are working on a "programming language for privacy" called Fuzz
  - Bob writes question in our language & submits it to Alice
  - Alice runs the program through our Fuzz system
  - Fuzz tells Alice whether it is okay to respond...
  - ... as well as a safe answer (including just enough noise)

How does Fuzz do this?

$$\frac{r \geq 1}{\Gamma, x :_r \tau \vdash x : \tau}\,var \qquad \frac{\Delta \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Delta + \Gamma \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}\,\otimes I$$

$$\frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Delta, x :_r \tau_1, y :_r \tau_2 \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}(x, y) = e\,\mathbf{in}\,e' : \tau'}\,\otimes E$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2}\,\&I \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i}\,\&E$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \begin{array}{c} \Delta, x :_r \tau_1 \vdash e_1 : \tau' \\ \Delta, x :_r \tau_2 \vdash e_2 : \tau' \end{array}}{\Delta + r\Gamma \vdash \mathbf{case}\,e\,\mathbf{of}\,x.e_1 \mid x.e_2 : \tau'}\,+E$$

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \mathbf{inj}_i\, e : \tau_1 + \tau_2}\,+I \qquad \frac{\Gamma, x :_1 \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \multimap \tau'}\,\multimap I$$

$$\frac{\Delta \vdash e_1 : \tau \multimap \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \Gamma \vdash e_1\, e_2 : \tau'}\,\multimap E \qquad \frac{\Gamma, x :_\infty \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\,\to I$$

$$\frac{\Delta \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \infty\Gamma \vdash e_1\, e_2 : \tau'}\,\to E \qquad \frac{\Gamma \vdash e : \tau}{s\Gamma \vdash\, !e : !_s\tau}\,!I$$

$$\frac{\Gamma \vdash e : !_s\tau \qquad \Delta, x :_{rs} \tau \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}\,!x = e\,\mathbf{in}\,e' : \tau'}\,!E \qquad \frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau}{\Gamma \vdash \underset{\mu\alpha.\tau}{\mathbf{fold}}\, e : \tau}\,\mu I$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \underset{\mu\alpha.\tau}{\mathbf{unfold}}\, e : [\mu\alpha.\tau/\alpha]\tau}\,\mu E$$

- Fuzz uses a type system to infer the relevant property (sensitivity) of a given query
  - If program typechecks, we have a proof that running it won't compromise privacy
  - Solid formal guarantee - no more accidental privacy leaks!

Intuition behind the type system

$$\begin{tikzcd}
\begin{array}{c} f(x) = x \\ \text{Sensitivity } 1 \end{array} \arrow[d] & & \begin{array}{c} f(x) = 7 \\ \text{Sensitivity } 0 \end{array} \arrow[ddl] \\
\begin{array}{c} f(x) = 2*x \\ \text{Sensitivity } 2*1 \end{array} \arrow[dr] & & \\
& \begin{array}{c} f(x) = 2*x + 7 \\ \text{Sensitivity } 2*1 + 0 \end{array} &
\end{tikzcd}$$

- Suppose we have a function f(x)=2x+7
  - What is its sensitivity?
  - Intuitively 2: changing the input by 1 changes the output by 2

Current directions

- Type inference (!)
- Adding *dependent types* to express more precise constraints on behavior
  - E.g., the fact that the sensitivity of a private *k-means* algorithm depends on how many rounds of iteration you ask it to perform

Thank you!