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

Types *à la* Milner

Benjamin C. Pierce
University of Pennsylvania

April 2012

Type inference

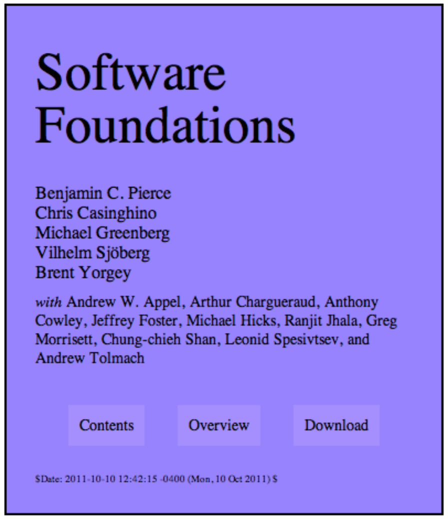
Abstract types

Types *à la* Milner

Types for interaction

(Types for differential privacy)

Milner and me



- Last ML postdoc at Edinburgh
  - and first-generation at Cambridge
- Happy ML user Unison
- Pi-calculus type systems (with Davide Sangiorgi, Dave Turner)
- Pict programming language (with Dave Turner)

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

- Local type inference → Scala
- POPLMark and Software Foundations

POPL mark

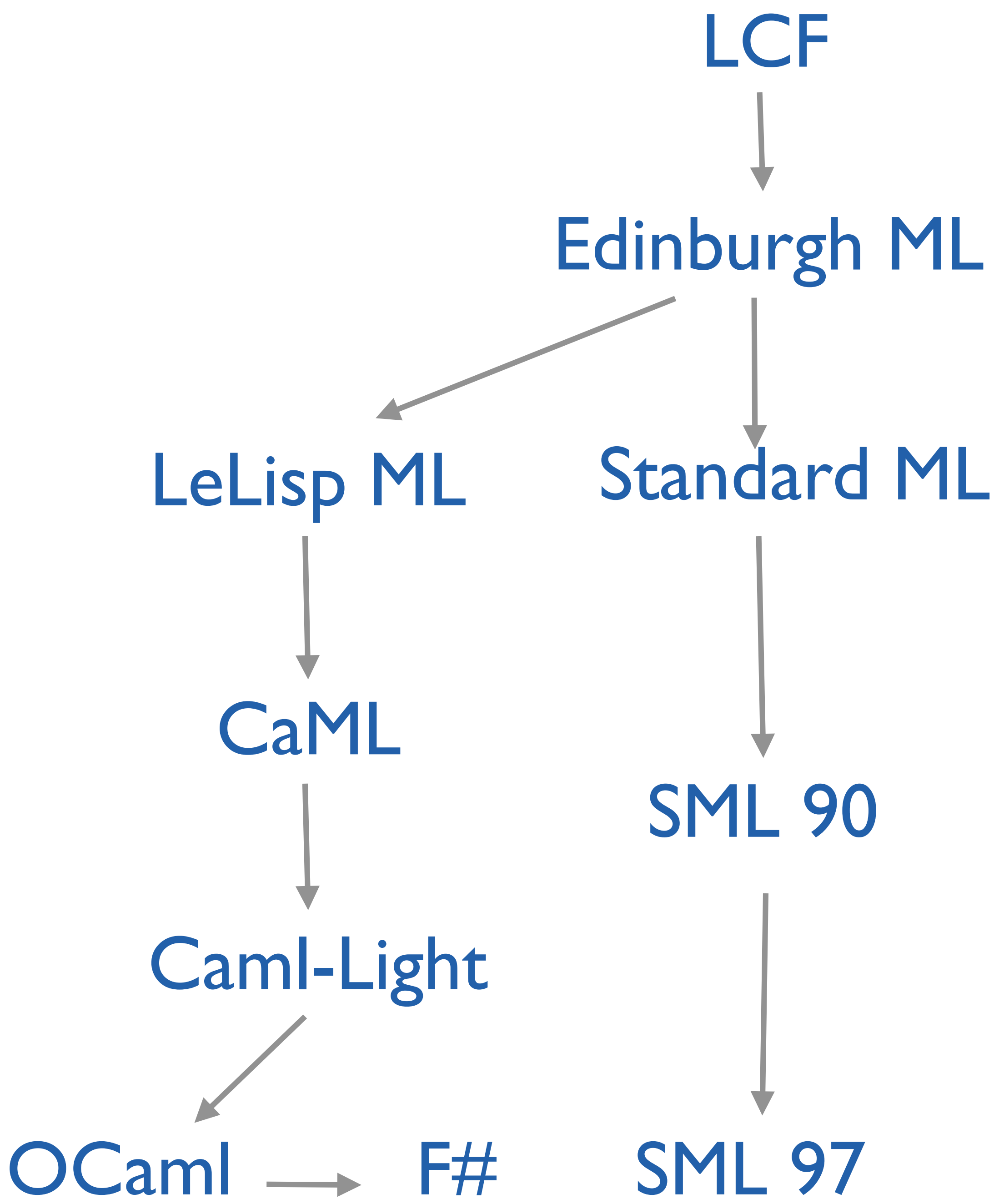
A Theory of Type Polymorphism in Programming

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.

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:

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

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)

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.

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

$$\begin{aligned}
\sigma_{map} &= \sigma_f \times \sigma_m \rightarrow \rho_1 \\
\sigma_{null} &= \sigma_m \rightarrow \textit{bool} \\
\sigma_{hd} &= \sigma_m \rightarrow \rho_2 \\
\sigma_{tl} &= \sigma_m \rightarrow \rho_3 \\
\sigma_f &= \rho_2 \rightarrow \rho_4 \\
\sigma_{map} &= \sigma_f \times \rho_3 \rightarrow \rho_5 \\
\sigma_{cons} &= \rho_4 \times \rho_5 \rightarrow \rho_6 \\
\sigma_{nil} &= \rho_6 \\
\rho_1 &= \rho_6
\end{aligned}$$

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

$$
\begin{aligned}
\sigma_{map} &= \sigma_f \times \sigma_m \to \rho_1 \\
\sigma_{null} &= \sigma_m \to \mathit{bool} \\
\sigma_{hd} &= \sigma_m \to \rho_2 \\
\sigma_{tl} &= \sigma_m \to \rho_3 \\
\sigma_f &= \rho_2 \to \rho_4 \\
\sigma_{map} &= \sigma_f \times \rho_3 \to \rho_5 \\
\sigma_{cons} &= \rho_4 \times \rho_5 \to \rho_6 \\
\sigma_{nil} &= \rho_6 \\
\rho_1 &= \rho_6
\end{aligned}
$$

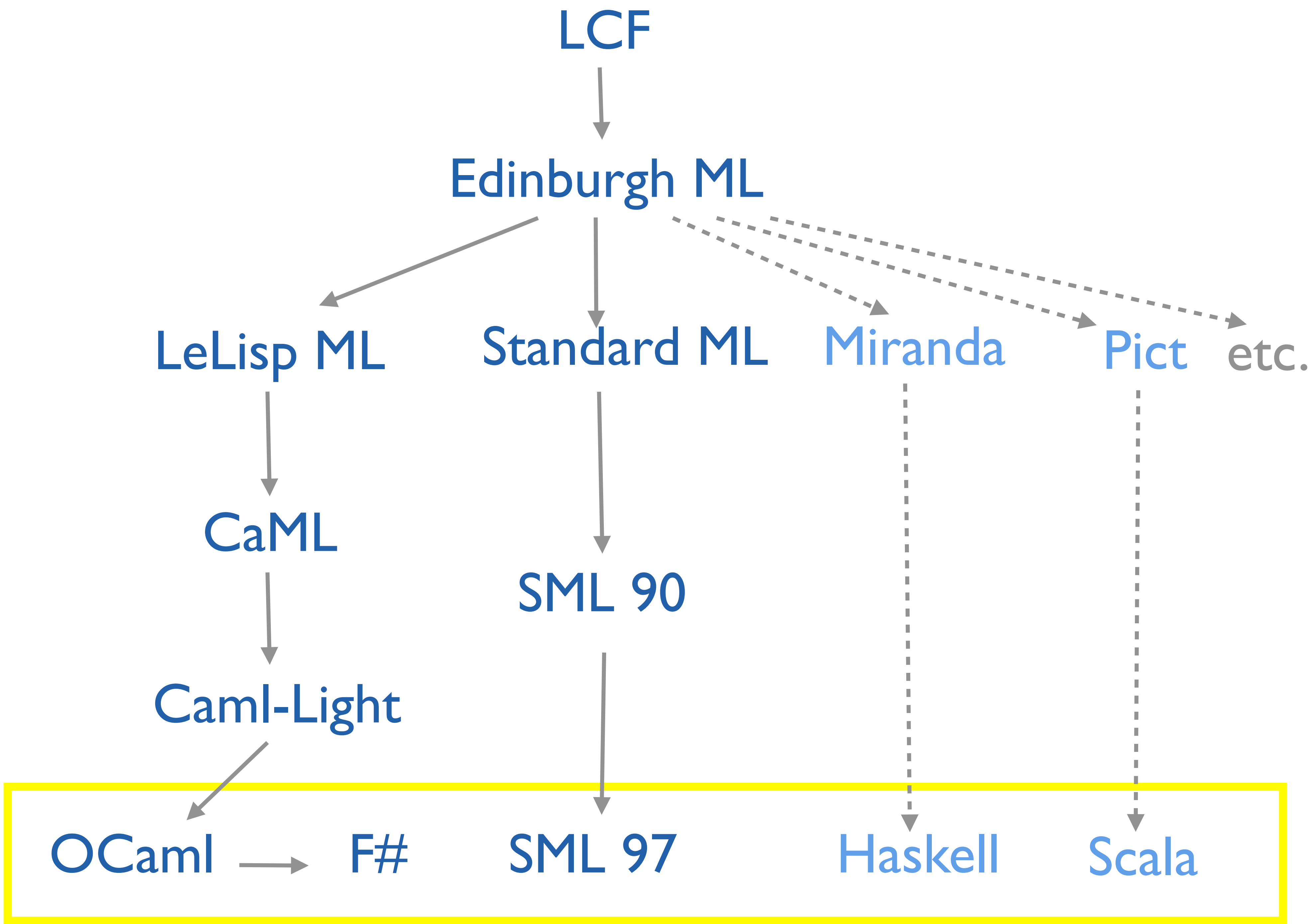
$$
\begin{aligned}
\sigma_{null} &= \alpha_1\ \mathit{list} \to \mathit{bool} \\
\sigma_{nil} &= \alpha_2\ \mathit{list} \\
\sigma_{hd} &= \alpha_3\ \mathit{list} \to \alpha_3 \\
\sigma_{tl} &= \alpha_4\ \mathit{list} \to \alpha_4\ \mathit{list} \\
\sigma_{cons} &= (\alpha_5 \times \alpha_5\ \mathit{list}) \to \alpha_5\ \mathit{list}
\end{aligned}
$$

Most general solution

$$\sigma_{map} = (\sigma_m \to \rho_4) \times \sigma_m\ \mathit{list} \to \rho_4\ \mathit{list}$$

Principal type

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


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

(P +Turner)

Local Type Inference

- Problem: How to combine
  - impredicative polymorphism
  - subtyping
  - type inference
- Idea: Abandon full type inference
  - just infer "locally best types" where possible
- When type arguments are omitted:
  - Compare actual and expected types of provided term arguments to yield a set of subtyping constraints on missing type arguments
  - Choose solution that satisfies these constraints while making the result type of the whole application as small (informative) as possible

Scala

What to call it?

37k google hits

Hindley-Milner?

13k hits

Damas-Milner?

4k hits

Damas-Hindley-Milner?

Milner's contribution

- Defined algorithm W
  - Generate a set of equational constraints from a program and use Robinson's unification algorithm to solve them
  - Generalize variables appropriately at let-bindings
- Proved soundness
  - Gave a (standard) denotational model for core ML
  - Showed that well-typed terms do not denote the special element *wrong* in the model
  - Showed that algorithm W finds some type for every well-typed term (and no ill-typed term)
- *Conjectured* completeness

Milner, *A Theory of Type Polymorphism in Programming*, 1978

Damas's contribution

- Proof of the completeness of Algorithm W
  - For every well-typed term, the algorithm finds a *principal type*, from which all other types for the term can be derived as instances

Damas and Milner, *Principal Type Schemes for Functional Programs*, 1982

Hindley's contribution

- Algorithm for inferring *principal type schemes* for terms in combinatory logic (S-K terms)
  - Also relied on Robinson's algorithm for solving equality constraints

Hindley, *The Principal Type-scheme of an Object in Combinatory Logic*, 1969

Curry’s contribution

- Independent proof of Hindley’s main result
  - ... but not relying directly on Robinson’s algorithm

... and don’t forget Morris ’68!

... or Newman ’43!

Curry, *Modified basic functionality in combinatory logic*, 1969

What to call it?

- Hindley-Milner (or Curry-Hindley-Milner-Morris-Newman!)
  - for unification-based type inference
- Milner
  - for the extension to let-polymorphism
- Damas-Milner
  - for the proof of completeness (principal types) for the let-polymorphism extension

Types in LCF

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).

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

Gordon, Milner, Morris, Newey, and Wadsworth, *A Metalanguage For Interactive Proof in LCF*, 1977

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 theorems

```
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!

Pi-calculus

P,Q	::=	0	inert process
		P | Q	P and Q in parallel
		!P	arbitrarily many copies of P in parallel
		$x?(y_1 \ldots y_n).\ P$	read $y_1 \ldots y_n$ from channel x and continue as P
		$x!(y_1 \ldots y_n).\ P$	send $y_1 \ldots y_n$ along channel x and continue as P
		$\nu x.\ P$	private channel x in P

$$(x!\ (y_1 \ldots y_n).\ P)\ |\ (x?\ (z_1 \ldots z_n).\ Q) \quad \Rightarrow \quad P\ |\ ([y_1 \ldots y_n / z_1 \ldots z_n]Q)$$

Milner's sort system

- Each channel is associated with a *subject sort*
- Each subject sort is associated with an *object sort*, which is a tuple of subject sorts
- A process is *well typed* if, at every send and receive, the object sort of the channel used for communication matches the subject sorts of the channels being sent or received

$$\{\,\mathrm{NAT} \mapsto (\mathrm{SUCC}, \mathrm{ZERO}),\ \mathrm{SUCC} \mapsto (\mathrm{NAT}),\ \mathrm{ZERO} \mapsto ()\,\}$$

$$\neq$$

$$\{\,\mathrm{NAT}` \mapsto (\mathrm{SUCC}`, \mathrm{ZERO}`),\ \mathrm{SUCC}` \mapsto (\mathrm{NAT}`),\ \mathrm{ZERO}` \mapsto ()\,\}$$

Milner, *The Polyadic Pi-Calculus: A Tutorial*, 1991

Structural types for pi

- associate each channel binder directly with a *type*
- make recursion explicit

T	::=	$ch(T_1 \ldots T_n)$	channel carrying $(T_1 \ldots T_n)$
		$\mu X.T$	recursive type
		X	type variable

$$\{\,\text{NAT} \mapsto (\text{SUCC}, \text{ZERO}),\ \text{SUCC} \mapsto (\text{NAT}),\ \text{ZERO} \mapsto (\,)\,\}$$

$$\mu X.\, ch(\, ch(X),\ ch()\,)$$

Polymorphic pi

(P + Sangiorgi)

- On each communication, pass a tuple of types *and* a tuple of channels
- Analogous to full 2nd-order lambda-calculus

T	::=	$ch(X_1 \ldots X_m, T_1 \ldots T_n)$	channel carrying types $(X_1 \ldots X_m)$ and channels $(T_1 \ldots T_n)$
		$\mu X.T$	recursive type
		X	type variable

e.g., ch(X, ch(X))

ch(X, Y, ch(X,ch(Y)), list X, list Y)

where list X = ch(ch(X), ch())

Pi + subtyping

(P + Sangiorgi)

- Separate *read* and *write* capabilities
- cf Reynolds's treatment of refs in Forsythe

T	::=	ch($T_1 \ldots T_n$)	read *and* write capabilities for channel carrying ($T_1 \ldots T_n$)
		in($T_1 \ldots T_n$)	read capability only
		out($T_1 \ldots T_n$)	write capability only
		…	

Linear pi

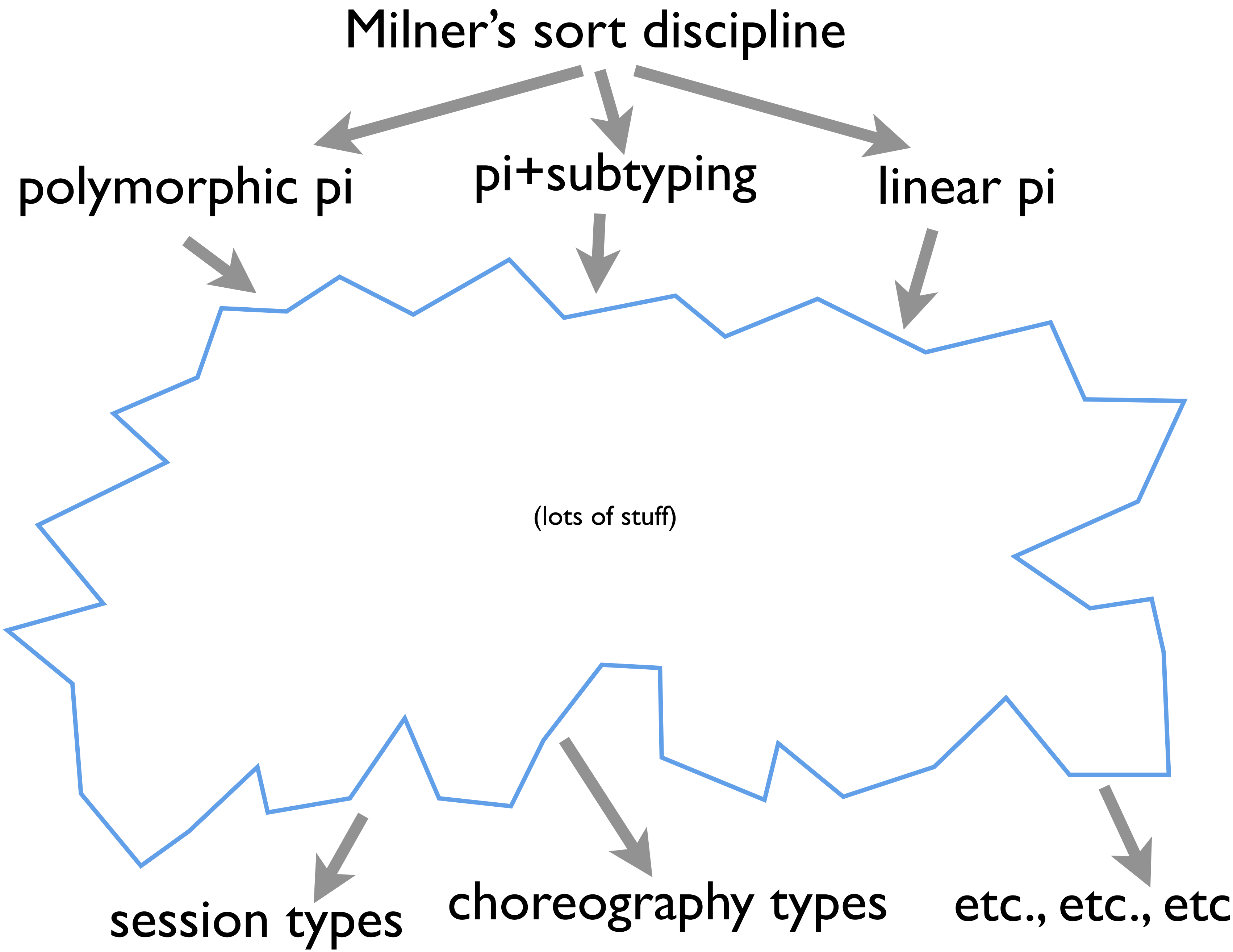
(Kobayashi, P, Turner)

- Track *use-once* capabilities
- cf. linear logic, linear lambda-calculi

T	::=	ch($T_1 \ldots T_n$)	ordinary channel
		ch!($T_1 \ldots T_n$)	use-once channel
		...	

Behavioral consequences

- Each of these refinements has interesting effects on behavioral equivalences
- E.g., in the pi-calculus with subtyping, we get stronger versions of standard theorems
  - e.g. a stronger *replicator theorem* than in the untyped language
- Validates beta-reduction for the pi-calculus encoding of CBV lambda-calculus
  - (not valid for untyped pi)


Milner's sort discipline
polymorphic pi
pi+subtyping
linear pi
(lots of stuff)
session types
choreography types
etc., etc., etc

Types for Privacy

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

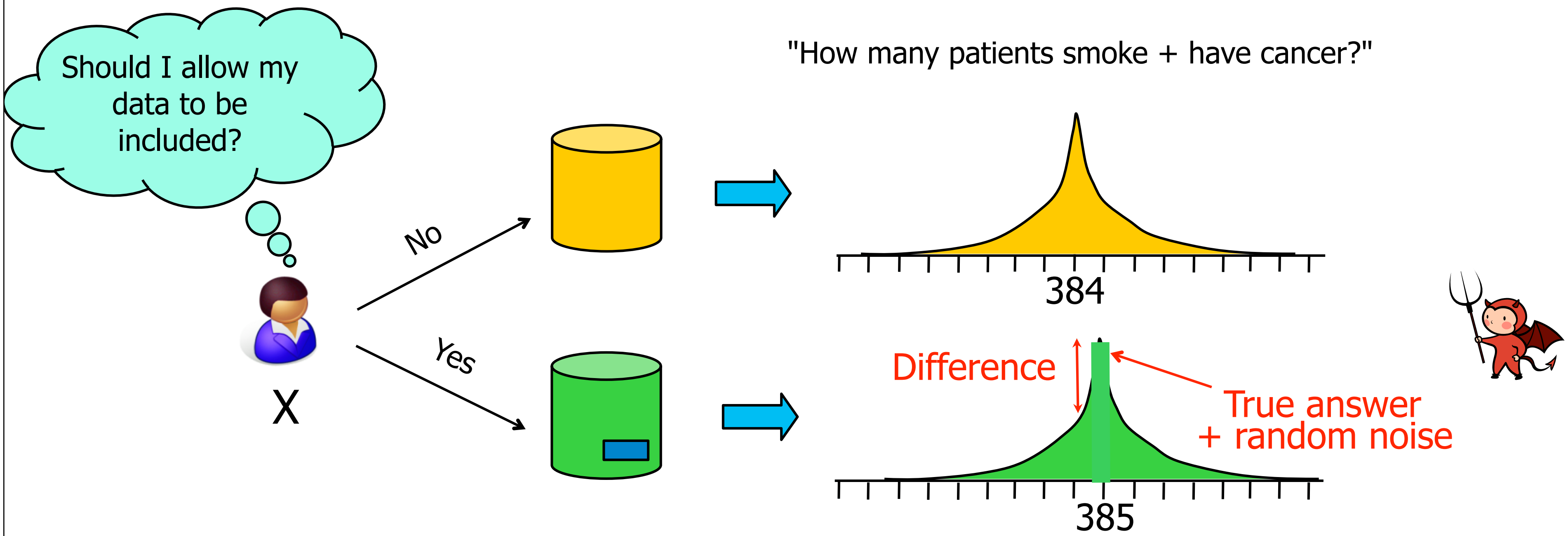
Motivation: querying private data

- 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

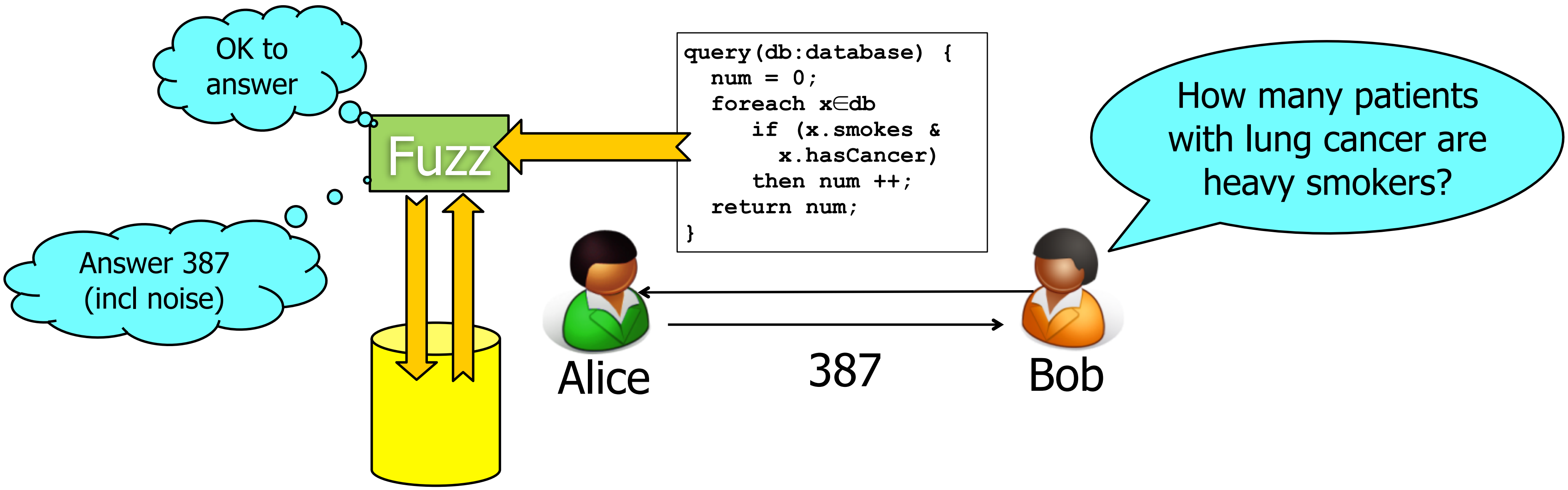


- Idea: Add a bit of noise to the answer
  - "387 patients smoke + have cancer, plus or minus 3"
- Can bound how much information is leaked
  - Even under worst-case assumptions!

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



- 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

f(x) = x
Sensitivity 1

f(x) = 7
Sensitivity 0

f(x) = 2*x
Sensitivity 2*1

f(x) = 2*x + 7
Sensitivity 2*1 + 0

- 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!