Types and Programming Languages
The Next Generation

Benjamin C. Pierce
University of Pennsylvania

LICS, 2003

Overview

Birds-eye view of what's happening in the world of types for programming languages (not logic or theorem proving)

Using 1993 and 2003 as reference points

Caveats

I'll be...

- painting with a broad brush
- painting with a lot of people's paints
- making a quick sketch, not a finished landscape

and giving few citations!

Overview

- Some changes in the PL world

Overview

- Some changes in the PL world
- Some mature research areas

Overview

- Some changes in the PL world
- Some mature research areas
- Some "still going strong" areas

Overview

- Some changes in the PL world
- Some mature research areas
- Some "still going strong" areas
- Some new kids on the block

Overview

- Some changes in the PL world
- Some mature research areas
- Some "still going strong" areas
- Some new kids on the block
- Some current trends and challenges

Some Big Changes

We've come a long way, baby

It doesn't always feel like it when you are working in the trenches, but the world of PL research has changed in some major ways in ten years...

The Java (and C#) Juggernaut

The "end of the argument" about safe languages, types, garbage collection, etc., etc.

PCC Hits the Big Time

A side effect of the success of Java and C# is that "proof carrying code" is already taking over as a standard format for exchanging and installing code over the net.

- Java and C# are simple instances (proofs ~ typing derivations or annotations that can be used to reconstruct them)
- much more ambitious variants are being proposed [foundational PCC, etc.]

Technology Transfers

Industrial interest $\longrightarrow$ shortening time-horizon for (at least some) technology transfer

- Transferring garbage collection into mainstream languages took $\sim$30 years
- Transferring F-bounded quantification and local type inference into Java (to form GJ) took $\sim$5 years

The security boom

At least some parts of the world (e.g. funding agencies) are waking up to the urgency of better software security.

Trivial link with PL: Can't build castles out of cardboard!

Less trivial: Maybe we can bring techniques and insights from programming languages to bear on understanding, formalizing, and checking "real security properties" (secrecy, authenticity, anonymity, etc., etc.).

Brought to you by the recent explosion and imminent collapse of the internet...

Rise of "Lightweight Formal Methods"

Don't prove correctness; just find bugs...

- Model checking
- "Light" specification and verification [ESC, SLAM, etc.] (not always even sound!)
- Typechecking!

The basic ideas are long-established; but industrial attitudes have been greatly "softened" by the success of model checking in hardware design.

> "Formal methods will never have any impact until they can be used by people that don't understand them."
>
> — (attributed to) Tom Melham

Big Type Systems

People are getting more courageous about working with complex type systems

- Surprisingly complex type systems are being used now in the real world (cf. GJ!)
- Very complex type systems are being explored in research

Type systems are also getting richer — being used to track stronger and stronger invariants on data objects and their assumptions about their environment

Pervasive Presence of Pi

The pi-calculus [Milner-Parrow-Walker] has had a huge influence on the PL research world.

- compelling in itself
  - first concurrent calculus to be both "powerful enough" (to encode lambda-calculus, e.g.) and also mathematically elegant and tractable
- popularized operational techniques such as bisimulation
- focused attention on name binding (ν)

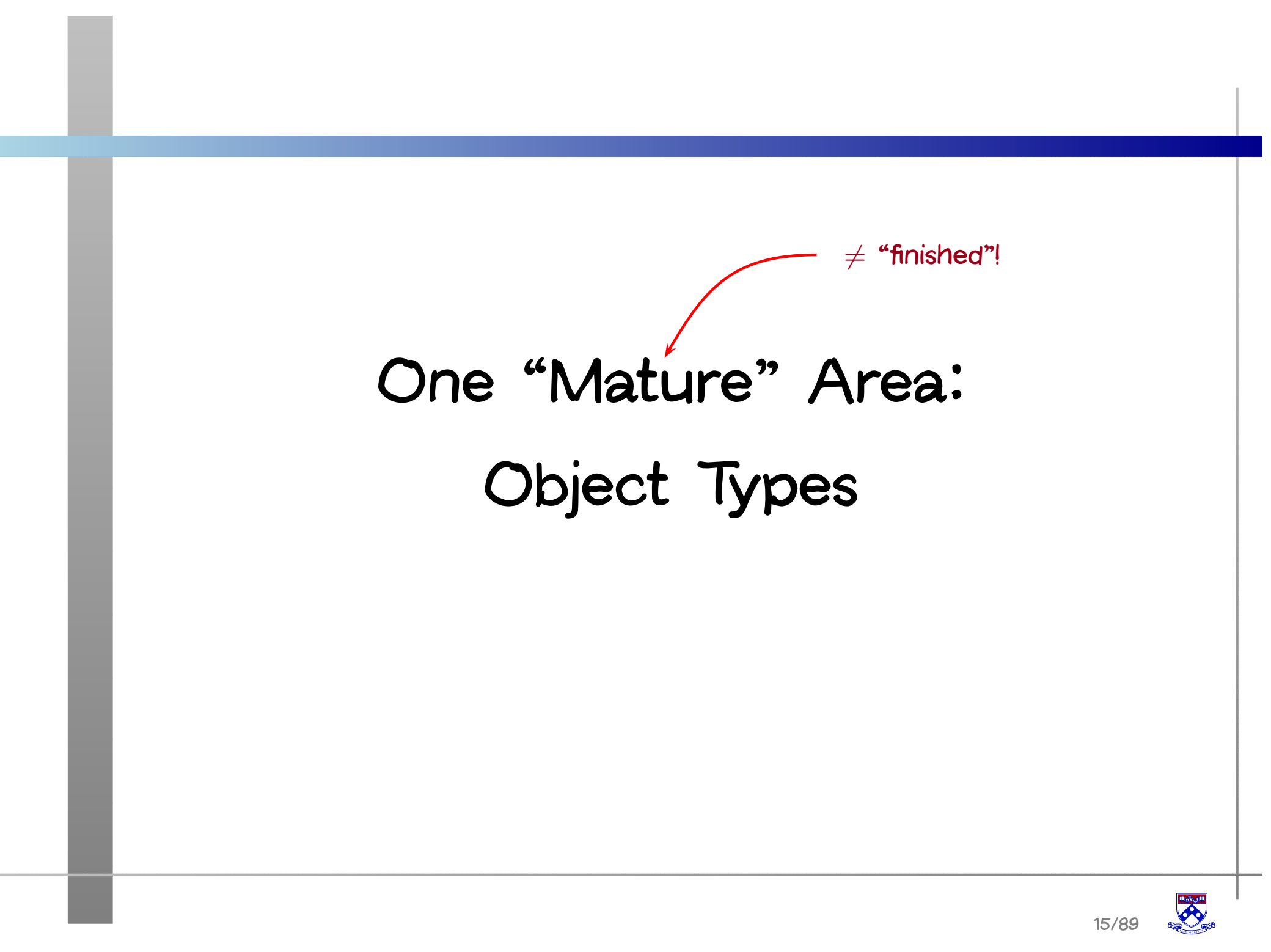
One result: Most PL people today know something about concurrency... a big change from ten years ago

Another result (?)...

Triumph of Operational Semantics

Since the early '90s, the focus in the PL community has moved from denotational descriptions and proof techniques to operational ones

- closer to PL practice (abstract machines)
- "lower-level" $\longrightarrow$ easier to deal with wide range of language features (in particular, concurrency!)
- increasingly powerful proof techniques becoming available [Pitts, etc.]



One "Mature" Area: Object Types

Overview

Goal: Formalize / explain core features of OO languages by "compilation" into some typed lambda-calculus

There is some disagreement about what are the core features, but people pretty much agree at least on:

- "OO-style" abstraction
- subtyping
- subclassing (or method override)
- open recursion through `self` (a.k.a. `this`)

Timeline

Prehistory Results Refinements Explotation

1983	A Semantics of Multiple Inheritance [Cardelli]
1989	Inheritance is not Subtyping [Cook, Hill, Canning]
1993	A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics [Bruce]
1993	Object-Oriented Programming Without Recursive Types [Pierce-Turner]
1994	A Theory of Objects [Abadi-Cardelli]
1996	On Binary Methods [Bruce-Cardelli-Castagna-HOG-Leavens-Pierce]
1996	An Interpretation of Objects and Object Types [Abadi-Cardelli-Viswanathan]
1997	Comparing Object Encodings [Bruce-Cardelli-Pierce]
1996	Objective ML: A simple object-oriented extension of ML [Rémy-Vouillon]
1998	Making the Future Safe for the Past: Adding Genericity to the Java Programming Language [Bracha-Odersky-Stoutamire-Wadler]
2001	Design and Implementation of Generics for the .NET Common Language Runtime [Kennedy-Syme]

OC vs. lambda-calculus encodings

The object calculus and lambda-calculus approaches are fundamentally very similar — they deal with the same range of phenomena and rely on essentially the same typing mechanisms.

However, the object calculus has proved more popular. Why?

- Immediacy: OC is obviously object-oriented $\longrightarrow$ appealing
- Implementability:
  - operational semantics of OC is close to something one could imagine implementing directly (though few people have!).
  - operational behavior of lambda-calculus encodings is easy to "get right modulo efficiency" but challenging to work out in a way that would satisfy an OO compiler writer
- Inflexibility: OC (in original form) does not support "depth subtyping" ($\{x:A,y:C\} \leq \{x:B,y:C\}$ if $A \leq B$)
  Leads to nice simplification of the theory
  Same restriction possible in λ-calculi, but "feels unnatural"

Technical Challenge

What is the precise relation between structural and nominal type systems?

- structural: type names are just abbreviations for (completely interchangeable with) their definitions; type equivalence, subtyping, etc. follow structure
  used in most research on type systems
- nominal: type names matter; subtyping declared by programmer (checked for consistency by compiler)
  used in most mainstream OO languages
  (GJ and C# generics are actually a complex hybrid)

Is there a general way to transfer mechanisms / results from one setting to the other?

Technical Challenge

What is the precise relation between structural and nominal type systems?

- structural: type names are just abbreviations for (completely interchangeable with) their definitions; type equivalence, subtyping, etc. follow structure
  used in most research on type systems
- nominal: type names matter; subtyping declared by programmer (checked for consistency by compiler)
  used in most mainstream OO languages
  (GJ and C# generics are actually a complex hybrid)

Is there a general way to transfer mechanisms / results from one setting to the other?

Cardelli-Gosling isomorphism, anyone?

Another Challenge

Explaining polymorphic OO programming to the masses

- golden opportunity for a good textbook! [your name here]

Some Ongoing Areas

Bounded Quantification

Basic Idea

Combine subtyping...

$$S \leq T$$

...and universal quantification...

```
All X. U
```

...with a twist:

```
All X<:T. U
```

Timeline

Basics

1985	On Understanding Types, Data Abstraction, and Polymorphism [Cardelli-Wegner] ("Kernel $F_{\leq}$")
1990–92	Coherence of Subsumption: Minimum typing and type-checking in $F_{\leq}$ [Curien-Ghelli] ("Full $F_{\leq}$")
1991	An Extension of System F with Subtyping [Cardelli-Martini-Mitchell-Scedrov]
1992	Bounded Quantification is Undecidable [Pierce]

Extensions

1993	Intersection Types and Bounded Polymorphism [Pierce]
1994	Subtyping in $F^{\omega}_{\wedge}$ is Decidable [Compagnoni]
1994	Higher-Order Subtyping [Pierce-Steffen]
1995	On Subtyping and Matching [Abadi-Cardelli]
1997	Termination of system F-bounded: A complete proof [Ghelli]
1999	Subtyping Recursive Types in Kernel Fun [Colazzo-Ghelli] LICS!

Type Inference

Motivation

The more interesting your types get, the less fun it is to write them down!

Trends

- For more powerful forms of polymorphism...
  - undecidability for System F [Wells] LICS!
  - for rank-2 polymorphism [Kfoury-Tiuryn] and intersections [Kfoury-Wells]
  - Type operators [Peyton-Jones, etc.]
- With constraints...
  - row variables [Wand, Remy]
  - subtyping [Aiken, Smith, etc.]
  - refinement types [Pfenning, Freeman, Davies, etc.]
  - type classes [Wadler-Blott, Jones]
  - HM(X) — a generic framework for type inference with let-polymorphism [Odersky-Sulzmann-Wehr]

Trends

- Partial...
  - based on higher-order unification [Boehm, Pfenning]
  - using datatype constructors as type annotations [Laufer-Odersky, Garrigue-Remy]
  - ML_F [Garrigue-Remy]
- Local...
  - Local Type Inference [Pierce-Turner]
  - Colored Local Type Inference [Odersky-Zenger-Zenger]

Effects

Idea

A type can describe not only the "shape" of the final result of a computation, but also (some approximation of) the effects it causes while evaluating

Timeline

Prehistory

Formalization

Exploitation

1987	FX-87 Reference Manual [Gifford-Jouvelot-Lucasses-Sheldon]

1992	Algebraic Reconstruction of Types and Effects [Jouvelot-Gifford]
1992	The type and effects discipline [Talpin-Jouvelot] LICS!
1994	Implementing the Call-By-Value Lambda-Calculus using a Stack of Regions [Tofte-Talpin]

---

mid '90s - now	ML Kit compiler
	Cyclone
	Cryptyc
	exception analyses
	process type systems with effects
	etc., etc.

Dependent Types

Trends

Recent work on dependent types in programming languages can be roughly divided into two streams:

- designing languages with full dependent types (a.k.a. "doing it the hard way") — e.g., Cayenne
- controlling dependent types to ensure tractable typechecking (and good interaction with nontermination, effects, etc.) — e.g., Pfenning and Xi

Timeline

Prehistory

1986 Typechecking Dependent Types and Subtypes [Cardelli]
1988 Phase distinctions in type theory [Cardelli]

New Exploration

1992 Pattern Matching with Dependent Types [Coquand]
1996 Subtyping Dependent Types [Aspinall-Compagnoni] LICS!
1998 Cayenne — a language with dependent types [Augustsson]
1999 Dependent types in practical programming [Xi-Pfenning]
1999 Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming [Hughes-Pareto]
and many follow-on papers on types for space-bounded computations
2003 A Nominal Theory of Objects with Dependent Types [Odersky-Cremet-Rockl-Zenger]

Also...

Module Systems

Overview

1993: Definition of Standard ML

- Powerful module system
- Unsatisfactory (clunky, implementation-oriented) formalization

2003: Type-theoretic account of SML-like module systems

Work in this decade has focused on developing type-theoretic foundations for...

- modules with types (definitions / singletons)
- controlled information hiding (translucency)
- parameterization (functors)
- "sharing"
- hierarchy (sub-structures)

Overview

At the same time, some progress has been made on formalizing alternative modularity ideas from elsewhere — in particular, from the OO community

- virtual types (objects with "type members")
- "mixin modules"

Timeline

Prehistory

1986 Using dependent types to express modular structure [MacQueen]

Foundations

1990 Higher-Order Modules and the Phase Distinction [Harper-Mitchell-Moggi]

1994 A Type-Theoretic Approach to Higher-Order Modules with Sharing [Harper-Lillibridge] ("translucent sums")

1994 Manifest Types, Modules, and Separate Compilation [Leroy]

1996 Mixin Modules [Duggan-Sourelis]

1997 Program Fragments, Linking, and Modularization [Cardelli]

1999 Non-dependent Types for Standard ML Modules [Russo]

Consolidation

2002 A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics [Ancona-Zucca]

2003 A type system for higher-order modules [Dreyer-Harper-Crary]

Challenge

How can these threads of work be brought together?

In particular, can we combine the benefits of...?

- ML-style modules (functors, sharing, etc.)
- objects, classes, and inheritance
- Haskell-style classes (overloading)
- mixins

Challenge: Pragmatics (ML-style)

Are the benefits of ML-style module systems really worth the costs (in particular, the hit to the language complexity budget)?

My own take:

- the costs are real and important
- the benefits are also real, but it is surprisingly difficult to come up with examples where there is *no* other way of doing things (using objects instead of modules, playing games with makefiles and linkers, accepting an occasional run-time type test, etc., etc.)

Can these benefits be explained to real programmers?

Are there other ways?

Challenge: Recursive Modules

Often-requested feature: mutually recursive modules (i.e., type and value recursion across module boundaries)
Surprisingly tricky in the context of the other features of ML-like languages:

- semantics of module recursion tricky to define in a call-by-value setting
- combining recursion with abstract types
- ensuring conservativity over the core language

Challenge

Strong module systems in dynamic settings

- marshalling (tricky when abstract types are involved) [Sewell et al.]
- dynamic loading
- dynamic re-loading [Hicks]

Some New Areas

Linear Types

Linear Types

Original idea: Linear types can be used to eliminate garbage collection

- true in some sense, but not useful — only works for exponential-free programs! [Chiramar, Gunter, Riecke]

Linear Types

Original idea: Linear types can be used to eliminate garbage collection

- true in some sense, but not useful — only works for exponential-free programs! [Chiramar, Gunter, Riecke]

Better idea: Linear types can be used to track many sorts of "capabilities" (and "obligations") in programs

- storage initialization in low-level programs
- deadlock prevention in concurrent languages
- alias types; islands; ownership types; etc.

Example: Vault

C dialect with a linear type system for managing resources such as memory blocks, files, network connections, graphics contexts, etc.
Basic ideas:

- Annotate types with keys representing capabilities of various sorts
- Track set of "held keys" in each computation state
  - e.g., opening a file creates a key
  - closing requires a key and destroys it
- Allow access to a resource only when all the keys associated with its type are currently held

Many refinements needed to make all this work in practice!

Process Types

Overview

The past decade has seen the establishment of a small industry in type systems for concurrent calculi such as the pi-calculus.

First steps: "Transplanted" type systems based on familiar typed lambda-calculi

Later developments: "Native" concurrent type systems incorporating notions of temporal or causal dependency.

Core ideas:

- modes (e.g., read-only, write-only, read-write channels)
- multiplicities (e.g., linearity)
- temporal ordering of interactions (natural outgrowth of linearity)

Timeline

Prehistory
Transplants
Going native
Exploitation

1974	The specification of process synchronization by path expressions [Campbell-Habermann]
70s–90s	Many type analyses for concurrent languages [Nierstrasz, Puntigam, etc., etc.]
1993	Typing and subtyping for mobile processes [Pierce-Sangiorgi] LICS!
1996	Linearity and the pi-calculus [Kobayashi-Pierce-Turner]
1997	Behavioral equivalence in the polymorphic pi-calculus [Pierce-Sangiorgi]
1996	Graph types for monadic mobile processes [Yoshida]
1997	A Partially Deadlock-Free Typed Process Calculus [Kobayashi]
1998	Language primitives and type disciplines for structured communication-based programming [Honda-Vasconcelos-Kubo]
1998–now	Many behavioral type systems for pi, ambient, join, etc. [Kobayashi, Yoshida, Honda, Igarashi, Hennessy, Reily, Sumii, Cardelli, Gordon, etc., etc.]
2001	A generic type system for the pi-calculus [Igarashi-Kobayashi]
2001–3	MSR Behave! project [Larus-Rajamani-Rehof]
2002	Resource Usage Types [Igarashi-Kobayashi]

Challenges

- Taming complexity! (Type inference helps, but not enough...)
- Developing analyses that are accurate enough in the presence of destructive update (needed for practical applications to mainstream imperative languages)
- Stress-testing in practice [Behave!]
- Folding new ideas back into sequential languages

Behave!

Type checkers are more concrete and model checkers are getting more symbolic.

Can they meet in the middle?

"Types as models"

Security Types

Security Types

Many intriguing and fruitful connections can be found between programming languages and mainstream security.

- non-interference $\longleftrightarrow$ contextual equivalence (bisimulation, etc.) [spi-calculus, ...]
- keys / nonces $\longleftrightarrow$ variable binding (references, channels, pi-calculus ν operator)
- etc.

Timeline

1982 Security Policies and Security Models [Goguen-Meseguer]
secrecy ∼ non-interference

1978 Syntactic Control of Interference [Reynolds]

1996 A Sound Type System for Secure Flow Analysis [Volpano-Smith]
security types for simple imperative language

1997 Secrecy by Typing in Security Protocols [Abadi]
"Un" types for security against untyped attackers (in spi-calculus)

1998 The SLam Calculus: Programming with Secrecy and Integrity
[Heintze-Riecke]

2001 Secrecy Types for Asymmetric Communication [Abadi-Blanchet]

2001 Authenticity by Typing for Security Protocols [Gordon-Jeffrey]
(Cryptyc project)

Challenges

Can we loosen the overly paranoid ("absolute secrecy is the minimum acceptable") demands of non-interference?

e.g.:

- relative secrecy [Volpano-Smith]
- declassification [Zdancewic]

Challenges

"Cryptographic parametricity"?

- Cryptography is a mechanism for information hiding

Challenges

"Cryptographic parametricity"?

- Cryptography is a mechanism for information hiding
- Polymorphism and abstract types are mechanisms for information hiding

Challenges

"Cryptographic parametricity"?

- Cryptography is a mechanism for information hiding
- Polymorphism and abstract types are mechanisms for information hiding
- Can a precise connection be drawn between them?

Challenges

"Cryptographic parametricity"?

- Cryptography is a mechanism for information hiding
- Polymorphism and abstract types are mechanisms for information hiding
- Can a precise connection be drawn between them?

E.g.:

- Is there a fully abstract translation from System F to an untyped lambda-calculus enriched with cryptographic primitives?
  [cf. current work by Sumii]

Challenges

Applying PL techniques to formalizing other concepts from the security literature (authenticity, anonymity, ...)

Example: Cryptyc

- domain-specific language (based on spi-calculus) for describing cryptographic protocols
  - "correspondence assertions" formalize authenticity properties
- domain-specific type system incorporating
  - channel types
  - effects
  - nonce types
  - 'untrusted type" for information from attackers [Abadi]

cf. Authenticity by Typing for Security Protocols [Gordon-Jeffrey]

High-level types for low-level languages

Overview

1996-98 were watershed years, with two highly visible developments (proof-carrying code and typed assembly language) drawing together earlier ideas into attractive bundles and energizing a great deal of new work.

"When bad languages do good types..."

Timeline

1960s-90s	various work on machine-level verification [e.g. Hoare]
1994	A type-based compiler for Standard ML [Shao-Appel]
1995	TIL: A type-directed optimizing compiler for ML [Tarditi-Morrisett-Cheng-Stone-Harper-Lee]
1996	Comparing Object Encodings [Bruce-Cardelli-Pierce]
1996-7	Proof-Carrying Code [Necula-Lee]
1998	From System F to Typed Assembly Language [Morrisett-Walker-Crary-Glew]
2001	Foundational Proof-Carrying Code [Appel (also Felty, Shao, etc.)] (and many other refinements and extensions of basic PCC and TAL)
2002	Cyclone: A Safe Dialect of C [Jim, Morrisett, et al.]
2002	CCured: type-safe retrofitting of legacy code [Necula et al]
2001	Enforcing High-Level Protocols in Low-Level Software (Vault project) [Deline-Fähndrich]

Types for XML

Goal

Motivation:

- XML documents often come with schemas describing their structure
- However, XML-processing languages either...
  1. ignore this structure, treating all XML documents uniformly as generic trees... (unsafe!)
  2. or translate schemas (and documents) into rough equivalents expressible using the language's native types and values ["data binding"]... (safe, but awkward!)

Goal: Develop a statically typed language with native support for XML

"Taking schemas seriously as types..."

Key Ingredients

There are several common schema languages for XML (DTD, XML-Schema, Relax-NG, ...). All are based on some form of regular tree automata.

In terms of types, we need:

- recursive types (regular trees)
- non-disjoint unions (non-determinism)
- subtyping (language inclusion)

Timeline

Theory

Languages

1980s	intersection types [Coppo-Dezani]
1988	Forsythe [Reynolds]
1991	recursive subtyping [Amadio-Cardelli]
1990s	union types [Dezani et al., Aiken et al., Church project]
2000	XDuce [Hosoya-Pierce-Vouillon]
early 2000s	XQuery [Fernandez, Simeon, Wadler, et al.]
2002	CDuce [Benzaken, Castagna, & Frisch]
2003	Xtatic [Gapeyev, Levin, Pierce Schmitt, etc.]

Example: XDuce

Goals:

- Demonstrate viability of native XML processing in a statically typed setting
- Develop fundamental theory and algorithms for "regular" types and pattern matching

Regular Types

`T`	::=	`String`	leaf
		`X`	type name
		`()`	empty sequence
		`T,T`	concatenation
		`l[T]`	tree labeled `l`
		`~[T]`	tree labeled anything
		`T|T`	union

Fix global set of (mutually recursive) definitions `X = T`.

Recursive uses of variables only allowed in rightmost positions and under labels (to keep things regular).

Standard regex operators (`T?`, `T*`, etc.) definable

Example: Types

```
type Addrbook = addrbook[Person*]
type Person   = person[Name,Email*,Tel?]
type Name     = name[String]
type Email    = email[String]
type Tel      = tel[String]

val mybook = addrbook[person[name["Haruo Hosoya"],
                             email["hahosoya@upenn"],
                             email["haruo@u-tokyo"]],
                      person[name["Jerome Vouillon"],
                             email["vouillon@upenn"],
                             tel["215-123-4567"]]]
```


Regular Patterns

Regular types suggest an elegant pattern-matching mechanism...

- statically typed "tree-grep"
- typechecker can do standard tests for exhaustiveness and irredundancy
- includes all of ML-style "algebraic pattern matching" as a special case
- experience in XDuce: very pleasant for programming

Regular Patterns

Regular patterns are just regular types decorated with variable bindings:

`P`	::=	`String`	leaf
		`X`	pattern name
		`()`	empty sequence
		`P,P`	concatenation
		`l[T]`	tree labeled `l`
		`~[T]`	tree labeled anything
		`P|P`	alternation
		`P as x`	binding

Linearity: The sub-patterns $\texttt{P}_1$ and $\texttt{P}_2$ in $\texttt{P}_1\texttt{|P}_2$ must bind the same set of variables. In $\texttt{P}_1\texttt{,P}_2$ they bind disjoint sets.

Example: A simple pattern match

```
match p with
   person[name[n], Email*, tel[t]]
     ->  (* do some stuff involving n and t *)
 | person[p]
     ->  (* do other stuff *)
```

Note how the type `Email*` is used in the first pattern to match a variable-length sequence of `email` nodes.

Example: A complete XDuce function

```
fun tels : Person* -> (Name,Tel)* =
   person[name[n], Email*, tel[t]], rest
                        ->  name[n], tel[t], tels(rest)
 | person[p], rest      ->  tels(rest)
 | ()                   ->  ()
```

A More Interesting Example

Using regular expression patterns, we can extract the subcomponents of an HTML table with a single `match`...

```
match t with
    table[cap as Caption?,
          col as (Col*|Colgroup*),
          hd as Thead,
          ft as Tfoot?,
          bd as (Tbody+|Tr+)]
      -> ...
```

Challenge: Integration with Objects

What was achieved in XDuce:

- basic definitions of regular types and pattern matching
- fundamental algorithms (subtyping, type-based pattern optimization)
- prototype implementation (good-quality front end + simple interpreter)

What was not achieved:

- High-performance pattern compilation
- Integration with other standard typing features — in particular, objects
- Inter-operability with established libraries and legacy systems

⟶ Xtatic

- Lightweight extension of C# with regular types and patterns

"Regular types for the masses"

Challenge: adding polymorphism

types	$\sim$	patterns
	$\Downarrow$	
polymorphic types	$\sim$	parameterized patterns

Writing down the definition of a system with regular types and polymorphism is not so hard.

Finding reasonable algorithms for deciding the subtype relation for such a system appears to be quite challenging.

Modal Types

Goal

Extract new typing ideas from (intuitionistic variants of) standard modal logics

Potential applications include type systems for...

- run-time code generation
- meta-programming and higher-order syntax with free-variables
- memoization and incremental computation
- information flow and security
- distributed computation
- resource-bounded computation
- etc., etc.

Timeline

Enabling

1993 The Proof Theory and Semantics of Intuitionistic Modal Logic [Simpson]

2001 Categorical and Kripke Semantics for Constructive Modal Logics [Alechina-dePaiva-Mendler-Ritter]
and other papers by same authors

Exploration

1996 A Temporal Logic Approach to Binding-Time Analysis [Davies] LICS!

2001 A Judgmental Reconstruction of Modal Logic [Davies-Pfenning]

2001 A modal analysis of staged computation [Davies-Pfenning]

Language design

1997 [Taha-Sheard] (MetaML)
and many follow-on papers

Polytypic Programming

(or Generic Programming)

Goals

Use type analysis for...

- eliminating boilerplate code in programs that "walk over" complex data structures
- efficient (tag-free) compilation of polymorphic code
- dynamic type-testing

Trends

- What information is used at runtime?
  - none (completely static specialization) (liked by the polytypic people b/c no run-time cost. Has limits in expressiveness though.)
  - dictionary-passing (Haskell type classes)
  - type passing
- What types can be analyzed?
  - atomic types of kind *, e.g. `int`, `char` (prehistory, overloading)
  - atomic types of any kind (constructor classes [Jones])
  - inductively defined types (intensional polymorphism, extensional polymorphism)
  - type constructors of kind `* -> *` [PolyP]
  - type constructors of kind `* -> ... * -> *` [Functorial ML]
  - type constructors of any kind [Hinze]
- Type-level type analysis?
  - intensional polymorphism
  - starting to show up in Generic Haskell

Challenge

Getting a good handle on reflection (as found in Java, etc.)

Finishing Up...

Summary

- "mature"
  - object types
- "ongoing"
  - bounded quantification
  - type inference
  - effects
  - dependent types
  - module systems
- new (or newly energized)
  - linear types
  - process types
  - security types
  - types for low-level languages
  - types for XML
  - modal types
  - polytypic programming

...and lots of other stuff!!

A couple last challenges

New Influences from Semantics?

E.g.:

- Separation logic [O'Hearn, Reynolds, etc.] has made impressive progress in reasoning about heap-manipulating programs

  What would a type system based on separation logic look like? What could it be used for?

- What about game semantics? (Could a meta-language for strategies somehow be useful for programming?)

- etc.

Mechanization

We are getting really, really, REALLY tired of writing and reading subject-reduction proofs.

When are we going to have tools that let us formalize all this stuff conveniently?

> The next functional language should have a completely mechanized abstract syntax, elaborator, and semantics with a completely machine-checked proof of type soundness.
>
> — Greg Morrisett

Acknowledgements

Many thanks for all the great conversations that went into the making of this talk!

Thorsten Altenkirch
Bob Harper
Naoki Kobayashi
Xavier Leroy
Alan Jeffrey
Greg Morrisett
Aleks Nanevski
Andy Pitts
Frank Pfenning
Francois Pottier
Didier Rémy
John Reynolds
Andre Scedrov
Alan Schmitt
Eijiro Sumii
Stephanie Weirich
Steve Zdancewic

...and everybody else that's taught me about types over the years