Types

Benjamin C. Pierce
University of Pennsylvania

Programming Languages
Mentoring Workshop, Jan. 2012

What are types?

Many definitions out there,
some of them useful...

type system

$\approx$

the part of a compiler that tells you it doesn't understand your program type system

$\approx$

a linguistic mechanism
for reasoning about
program behavior

What's good about types?

They are the world's best
lightweight formal method!

"Formal methods will never have a significant impact until they can be used by people that don't understand them."

- (attributed to) Tom Melham

Why I like working on types...

Poster-child for PL research

- Lightweight formal method ➜ big real-world impact
- Industrial uptake of research results
- E.g., Java, C#, Scala, ...

Beautiful theory + interesting engineering

- Prove theorems on Monday and Wednesday, hack on Tuesday and Thursday...

Connections to *many* other areas

- Logic
- Automata theory
- Compilers, static analysis
- Language design
- Formal methods, automated verification, ...
- Security, databases, systems, ...

Excellent colleagues :-)

Where to start

- Attend POPL   :-)
- Useful background courses:
  - Logic (as much as possible)
  - Algebra
  - Automata theory
  - Compilers
  - Computer architecture
- Some useful books:
  - Robert Harper, *Practical Foundations for Programming Languages* (manuscript, 2012)
  - Benjamin C. Pierce *et al.*, *Software Foundations* (electronic textbook, 2012)
  - Benjamin C. Pierce, *Types and Programming Languages* (MIT Press, 2001)
  - John Mitchell, *Foundations for Programming Languages* (MIT Press, 1996)

huge area
theory+practice
connections to everything

$\searrow$

summary = impossible

A selection of upcoming POPL papers

Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq

Strub, Swamy, Fournet, and Chen

Themes

1. Bringing dependent types into practical programming
2. Typechecking with SMT solvers
3. Machine-checked metatheory

Dependent types

Dependent types arise when term-level expressions are allowed to appear in types:

$$\begin{aligned} \mathrm{nil} &\in \mathrm{Vec}[0] \\ \mathrm{cons} &\in \Pi n{:}\mathrm{Nat}.\ \mathrm{Elt} \to \mathrm{Vec}[n] \to \mathrm{Vec}[n+1] \end{aligned}$$

Such types can express extremely precise specifications of program behavior:

$$\begin{aligned} \mathrm{append} \in\ & \Pi m,n{:}\mathrm{Nat}. \\ & \mathrm{Vec}[m] \to \mathrm{Vec}[n] \to \mathrm{Vec}[m+n] \end{aligned}$$

Dependent types in practice

Dependent types have been used for decades in proof assistants based on constructive logic (Lego, Twelf, Coq, ...)

Incorporating them in practical programming languages has proved more challenging, but recent years have seen a flowering of such languages

- Cayenne (Augustsson 1998), ATS (Xi 2003), Epigram (McBride 2004), Aura (Jia et al. 2008), Fable (Swamyet al. 2008), F7 (Bengtson et al. 2008), Guru (Stump et al. 2008), Fine (Swamy et al. 2010), F* (Swamy et al. 2011), PCML5 (Avijit et al. 2010), Ur (Chlipala 2010b), Trellys (Casinghino et al. 2011)

Dependent types in practice

In particular, the F* language developed at Microsoft Research has been used to verify (by typechecking) over 30,000 LOC, including

- real security protocols
- web browser extensions
- distributed applications

Typechecking with SMT solvers

Another good idea:

- Typechecking with dependent types (and other rich typing disciplines) generates many proof obligations that require sophisticated logical inference to discharge
- Satisfiability-modulo-theories (SMT) solvers are astonishingly good at this sort of inference
- So: Typechecker generates obligations, encodes as logical propositions, and sends to a stock SMT solver
  - Pro: Enormous increase in expressiveness of types
  - Con: Loses completeness of typechecking, and some degree of predictability

Machine-checked metatheory

One more thread:

- Formal proofs of fundamental metatheorems (e.g. type safety) for full-scale PLs using proof assistants like Coq
- Not quite "routine," but many examples now
- POPLmark challenge [2005] helped energize the area

Problem

No formal connection between the idealized type system about which metatheorems are proved and its implementation in a concrete typechecker

Prior approaches

- Build a typechecker in a language of your choice and use your favorite program logic to prove it correct.
  - Not much fun
- Build a typechecker in Coq, prove it correct, and then extract it to ML or Haskell to get a high-performance implementation
  - Can be done — e.g., CompCert
  - But requires programming in Coq's rather restrictive language (no state or exceptions, all functions must be total, etc.)

This paper

1. Observe that, if we're implementing a dependently typed language L, we can build a typechecker for L *in itself!*
2. Now write a specification *as a type* S that says "This typechecker always builds well-formed certificates of typing."
3. Run the typechecker on itself to produce a certificate C that it says obeys the specification S.
4. Load this certificate into Coq and use the fact that L is sound (a theorem separately formalized in Coq) to prove that C is a true description of the typechecker's behavior.

Nested Refinements: A Logic For Duck Typing

Chugh, Rondon, and Jhala

Major trend

Bridging the gap between statically typed languages (ML, Haskell, Java, C#, ...) and “dynamic languages” (Python, Ruby, Lua, JS, etc.)

Many approaches:

- “soft typing”
- contracts
- gradual typing
- ...

Goal of this paper

Type inference for dynamic scripting languages

Challenge

Tricky combination of features!

- dynamic type tests (is-List, is-Function, ...)
- dictionaries (with computed keys)
- higher-order functions
  - and dictionaries of HO functions, aka objects

Technical foundation

Refinement types

$$\{ v : T \mid P(v) \}$$

set comprehensions

$\{ x \in S \mid P(x) \}$

in constructive type theory (Nordstrom and Petersson, '83); VDM (Jones '86); PVS [(Rushby, Owre, Shankar '78); ...

subset types

(in specification languages / theorem provers)

Meyer (Eiffel); Findler and Felleisen (ICFP 2002); ...

Freeman and Pfenning (PLDI '91); Xi and Pfenning (POPL '99); ...

contracts

(dynamically checked)

$\{ v : T \mid \text{pred } v \Downarrow \text{true} \}$

refinement types

(over decidable theories)

$\{ v : T \mid P(v) \}$

specialized decision procedure / dynamic checking ⇒ external SMT solver

Flanagan, Hybrid Type Checking (POPL '06); Bierman, Gordon, Hritcu, and Langworthy, Semantic subtyping with an SMT solver (ICFP, 2010)

Nested refinements

(this work)

Nested Refinements

- Intuition
    - Allow function types to appear (as uninterpreted atoms) in predicates
    - Interleave logical inference by the external SMT solver with syntactic subtyping for function types
- Major technical challenge: Soundness
    - Standard “substitution preserves typing” property broken!
    - Recover by proving soundness for a more general system

Duck Typing??

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

James Whitcomb Riley (1849–1916)

Duck Typing??

=

Typing?!

The Ins and Outs of Gradual Type Inference

Rastogi, Chaudhuri, and Hosmer

A Type Theory for Probability Density Functions

Bhat, Agarwal, Vuduc, and Gray

Background

Many papers have studied lambda-calculi with *probability distributions* as first-class values — e.g.

- Ramsey and Pfeffer. *Stochastic Lambda Calculus and Monads of Probability Distributions*. POPL '02.
- Park, Pfenning, and Thrun, *A Probabilistic Language based upon Sampling Functions*. POPL '05.

Applications:

- Machine learning
- Monte carlo methods
- Test data generation
- Differential privacy
- ... many more ...

Problem

- Discrete distributions are well supported
  - Finite or countable number of possible outcomes
- Continuous and hybrid distributions less so
  - "... no existing language rigorously supports expressing the probability density function (PDF) of custom probability distributions."
  - Required for many important statistical techniques

Contributions

- A core language for user-defined probability distributions on discrete, continuous, and hybrid probability spaces
    - Semantics defined using classical measure theory
- A type system for checking that a term denotes an absolutely continuous distribution — i.e., one that has a probability density function
- Procedure that calculates PDFs for a large class of well-typed distributions

Key Idea

- If T is a type, then dist T is the type of distributions over T
  - roughly, functions from T to [0,1] whose ranges sum to 1
- Probability distributions form a monad (that is, the type constructor dist is a monad)
  - Old idea (Lawvere, *The category of probabilistic mappings*, 1962; Claire Jones PhD thesis, 1990)

The probability monad

Two basic constructors:

- return ∈ T → dist T
  - distribution assigning probability 1 to v
- bind ∈ dist S → (S → dist T) → dist T
  - defines one distribution by sampling from another, a.k.a. conditional probability

Thank you!

Any questions?

(Come study at Penn! :-)