Type inference and modern type systems

Stephanie Weirich
University of Pennsylvania

Type inference

- What is the role of type inference in the design of programming languages?
- Old answer (for many FP languages): Part of the language specification
  - Defines what valid programs
  - *Disabling* technology for advanced type systems

A different philosophy

- Type inference as an afterthought:
  - Expressive (but wordy) language is the standard
  - Inference is a tool to help programmers produce programs, but no more
  - Other means of program generation may use different tools
- Allows more sophisticated type systems

Studying the tool

- But that doesn't mean we shouldn't study type inference
- Need a specification of the tool
- Opportunities for research into type system design
  - This talk: examples from AspectML, Haskell
  - My goal this week: more examples from type systems for program generation

Specific examples

- AspectML, aspect-oriented functional programming language
  - polymorphic aspects and first-class pointcuts
  - run-time type analysis
- Haskell
  - GADTs
  - Higher-rank/impredicative polymorphism

Trade off

- Unifying theme
  - All of these languages use typing annotations to augment HM type system
  - Each type system distinguishes between “known” and “guessed” type information, with more or less precision.
- Trade-off between simple specification and pithy language
- This talk: some details about AspectML, highlights of rest

AspectML

- Aspects = advice + pointcuts
- Pointcuts
  - “where” advice happens
  - Currently: sets of function names (other joinpoints possible)
- Advice
  - “what” happens before/after/around joinpoints

First-class pointcuts

- Code to install advice may be used for many different pointcuts
  - example: a “debugger” aspect can use a single function to install tracing code
- Pointcuts must have types
  - specify interface between joinpoints and advice
  - advice may be polymorphic

pc : pointcut (all a. a → Int)

advice before pc [a] (x:a, ...) { ... }

Polymorphic pointcut typing

- May assume joinpoint is more polymorphic

  let pc = { f } : pointcut (Int → Int)

  advice after pc [a] (x:a,...) { .... }

- Can't assume joinpoint is *less* polymorphic

  let pc = { g } : pointcut (a. a → a)

  advice before pc (x:int,...) { ... }

Pointcut typing

- Computing pointcut type requires *anti-unification*

$f : Int \rightarrow Int$

$g : a \rightarrow Int$

$let\ pc = \{\ f, g\ \} :\ pointcut\ (a.\ a \rightarrow Int)$

Typecase

```
let add_tracing (pc : pointcut (ab. a -> b)) =
 let print[a] (x:a) =
   (typecase[unit] a of
     int =>  print_int x
     bool =>  print_bool x
     (b,c) => print (fst x); print (snd x)
     (b->c) => print "<function>") in
  advice before pc [a] (x:a, f,s) {
   print x;   x
}
```

Type inference problems in Aspect ML

- Pointcuts
  - anti-unification/unification
  - Can’t guess pointcut type - like first-class polymorphism
  - Specification of HM let-polymorphism is too flexible, functions have multiple types
- Typecase
  - Most examples require polymorphic recursion
  - Can be difficult to determine result type
  - Pathological examples with no most-general type

Let polymorphism

Specification in HM:

$$\frac{\Gamma \vdash e : t \qquad Gen(\Gamma,t) = s \qquad \Gamma, x{:}s \vdash e' : t'}{\Gamma \vdash \text{let } x = e \text{ in } e' : t'}$$

Allows multiple derivations:

$$\frac{\begin{array}{c}\vdash \lambda x.x : int \to int \\ Gen(int \to int) = int \to int \\ f : int \to int \vdash f\ 3 : int\end{array}}{\vdash \text{let } f = \lambda x\ .\ x \text{ in } f\ 3 : int}$$

$$\frac{\begin{array}{c}\vdash \lambda x.x : a \to a \\ Gen(a \to a) = \forall a.\ a \to a \\ f : \forall a.\ a \to a \vdash f\ 3 : int\end{array}}{\vdash \text{let } f = \lambda x\ .\ x \text{ in } f\ 3 : int}$$

What is type of {f} ?

Pathological typecase

```
f  x = typecase a of
          int => 2 + x
```

- Does f have type
  - int -> int
  - forall a. a -> int
  - forall a. int -> a
  - forall a. a -> a
- Most general type is not expressible:
  - forall a. (a=int) => a -> a

Simple, fairly conservative solution

- Typing annotations resolve ambiguity
  - typecase
    - Annotate for polymorphic recursion
    - Annotate return type, variables with "refined" types
  - pointcuts
    - When created or used
    - When arguments to functions
- Typing spec distinguishes "known" and "inferred" types
  - Context distinguishes types of variables
- Investigating how well this simple spec works

GADTs

A small bit of typesafe metaprogramming

```
data Exp a where
  Lit    : a -> Exp a
  If     : Exp Bool -> Exp a -> Exp a
  App : Exp (a -> b) -> Exp a -> Exp b
  Plus : Exp (Int -> Int -> Int)

eval :: Exp a -> a
eval (Lit x)       = x
eval (If x y z)   = if (eval x) then (eval y) else (eval z)
eval (App x y)  = (eval x) (eval y)
eval (Plus) = +
```

Type inference and GADTs

- Similar problems as typecase
- Annotations more burdensome here
  - typecase
    - always know scrutinee from program text
    - not that common (?)
  - GADTs
    - may not know *type* of scrutinee
    - must generalize normal case analysis (and deal with nested case analysis, existentials, etc.)

Wobbly types

- GADT type system distinguishes between “wobbly” and “rigid” types.
  - Typing rules push rigid types into the judgment
  - A type is wobbly if *any* part of it must be guessed
- Special rules also propagate “rigidity” through polymorphic function application

Higher-rank / impredicative polymorphism

- Allows polymorphic values to be stored in data structures and passed to functions
- Example: polymorphic, reified code
  - Now : code(tau)
  - Allows: code(sigma)
- Is this useful in practice?
  - polymorphism in meta-language allows polymorphism in object language
  - forall a. code (a -> a) vs. code (forall a. a -> a)

Boxy types

- Most precise system: boxes in the type separate “known” and “guessed” type information
- Essential for specification of impredicative instantiation
- Annotations propagated throughout the typing judgment

For more information

- Papers available from my website
- AspectML
  - with David Walker, Geoff Washburn, Dan Dantas
- GADTs - wobbly types
  - with Simon Peyton Jones, Dimitrios Vytiniotis, Geoff Washburn
- Impredicativity/Higher rank - boxy types
  - with Simon Peyton Jones, Dimitrios Vytiniotis