How to think about types: Insights from a personal journey

Frank Pfenning

Department of Computer Science
Carnegie Mellon University

Programming Languages Mentoring Workshop
Lisbon, Portugal, January 15, 2019

"Type" is the most common word in the abstracts for all papers submitted to, accepted at, and rejected from POPL this year

Once Upon a Time . . .

- 1980–1986 Working on TPS, a theorem prover for higher-order logic, in Common Lisp
- 1986 Dana Scott and Bill Scherlis hire me as a postdoc for the ERGO project on semantically based programming
- 1986 Gérard Huet, Thierry Coquand, Christine Paulin visit CMU
  - Gérard Huet gives course on *Computation & Deduction* using CAML as a metalanguage

Once Upon a Time . . .

- 1980–1986 Working on TPS, a theorem prover for higher-order logic, in Common Lisp
- 1986 Dana Scott and Bill Scherlis hire me as a postdoc for the ERGO project on semantically based programming
- 1986 Gérard Huet, Thierry Coquand, Christine Paulin visit CMU
  - Gérard Huet gives course on *Computation & Deduction* using CAML as a metalanguage
- Discovered the joy of static typing!

The Joy of Static Typing

- Transition from Lisp to ML
  - Productivity++
  - Bugs−−
- Some reasons
  - Clearly express data representations
  - Elegant pattern matching
  - Avoiding gross latent bugs under program evolution
  - Enforced module boundaries (not just name spaces)

Evaluation axis: How much dynamic checking is required

Lesson: Types should be statically checked

Simple Types

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \tau_1 \to \tau_2 \mid \dots \\
\text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \mid \dots \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

$$
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{hyp}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ {\to}I
\qquad
\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ {\to}E
$$

Simple Types

$$\begin{array}{llll} \text{Types} & \tau & ::= & \tau_1 \to \tau_2 \mid \ldots \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \mid \ldots \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{hyp}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \to\! I \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1} \to\! E$$

- Dynamics: computation rules $e \mapsto e'$ and values v
- Should λ-expressions be $\lambda x{:}\tau.\, e$?

Theorem [Preservation]: *If* $e : \tau$ *and* $e \mapsto e'$ *then* $e' : \tau$

Theorem [Progress]: *If* $e : \tau$ *then either* e *is a value or* $e \mapsto e'$ *for some* e'

Running Example: Binary Numbers

- "Little Endian" representation

```
datatype bin =
    E                           (* E = 0 *)
  | B0 of bin                   (* B0(x) = 2*x *)
  | B1 of bin                   (* B1(x) = 2*x+1 *)

val zero = E
fun succ E = B1(E)
  | succ (B0(x)) = B1(x)
  | succ (B1(x)) = B0(succ x)
```

Lesson: Strive for simplicity and elegance

Issue: Missing Branches

```
(* pred(x+1) = x *)
fun pred (B0(x)) = B1(pred x)
  | pred (B1(x)) = B0(x)

(*
  binary.sml:11.5-12.25 Warning: match nonexhaustive
*)
```

- For larger pieces of code, a pervasive occurrence
  - Either a genuine oversight (missing branch)
  - Or a reflection of an invariant outside the type system
- A significant source of bugs!

Refinement Types

- Express more program properties
  - Increase precision
- Rule out more programs
  - Do **not** increase generality
- Layered architecture
  - Simple types for approximate checking
  - Refinement types (here: sorts) for further precision
  - Dependent refinements (indexed types) are another story

Example: Positive Binary Numbers

```
datatype bin = E | B0 of bin | B1 of bin
datasort pos =     B0 of pos | B1 of bin

val zero : bin
val zero = E

val succ : bin -> pos
fun succ E = B1(E)
  | succ (B0(x)) = B1(x)
  | succ (B1(x)) = B0(succ x)

val pred : pos -> bin
fun pred (B0(x)) = B1(pred x)
  | pred (B1(x)) = B0(x)
```

Subsorting

- Subsorting is a derived concept

  $\tau \leq \sigma$ *if a value of type* τ *is also a value of type* σ
- Infer for base sorts via tree automata inclusion

$$\text{pos} \leq \text{bin}$$

  Every positive number is also a binary number
- Extend to compound types "the usual way"

$$\text{bin} \to \text{pos} \leq \text{bin} \to \text{bin}$$

$$\text{bin} \to \text{bin} \leq \text{pos} \to \text{bin}$$

A Surprise: We Need Intersections

```
datatype bin = E | B0 of bin | B1 of bin
datasort pos =     B0 of pos | B1 of bin

val E : bin
val B0 : bin -> bin /\ pos -> pos
val B1 : bin -> bin /\ bin -> pos (* = bin -> pos *)
```

A Surprise: We Need Intersections

```
datatype bin = E | B0 of bin | B1 of bin
datasort pos =     B0 of pos | B1 of bin

val E : bin
val B0 : bin -> bin /\ pos -> pos
val B1 : bin -> bin /\ bin -> pos (* = bin -> pos *)
```

- Need intersection types!
- Type checking undecidable in general
- Refinement restriction makes inference decidable
- Algorithm is abstract interpretation

Key Rules

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e : \sigma}{\Gamma \vdash e : \tau \wedge \sigma} \wedge I$$

$$\frac{\Gamma \vdash e : \tau \wedge \sigma}{\Gamma \vdash e : \tau} \wedge E_1 \qquad \frac{\Gamma \vdash e : \tau \wedge \sigma}{\Gamma \vdash e : \sigma} \wedge E_2$$

- Combine properties of the same expression e
- Follows a similar style of introduction and eliminations
- Can infer subsorting for intersection types

$$\tau \wedge \sigma \leq \tau$$
$$\tau \wedge \sigma \leq \sigma$$

Evaluation axis: How precise is the type system?

Evaluation axis: How precise is the type system?

Lesson: Precision can be more important than generality

Lesson: Sometimes it is beneficial to extend a system further than anticipated

Lesson: Sometimes it is beneficial to extend a system further than anticipated

Lesson: Look around (intersections, tree automata, abstract interpretation)

Lesson: Sometimes it is beneficial to extend a system further than anticipated

Lesson: Look around (intersections, tree automata, abstract interpretation)

Lesson: Program, program, program

A Fly the Ointment

A Fly the Ointment

- Surprise: sort inference after type inference is practical!

A Fly the Ointment

- Surprise: sort inference after type inference is practical!
- Surprise: results of inference are difficult to understand and use
- Why?
  - Distance between location and source of error
  - Inference captures accidental properties of code

Trivialized Example

```
datatype bin = E | B0 of bin | B1 of bin
datasort pos =     B0 of pos | B1 of bin

fun pred (B0(x)) = B1(pred x)
  | pred (B1(x)) = B0(x)
  | pred E = diverge
```

- Infer pred : $bin \rightarrow bin \wedge pos \rightarrow bin$
- Might want to specify pred : $pos \rightarrow bin$
- Should be a sort error to write (pred e) unless e : pos

Bidirectional Type Checking

- How to live without full type inference
- Propagate type information bottom-up and top-down
  - But not haphazardly!
- Judgments
  - $\Gamma \vdash e \Leftarrow \tau$ (check e against τ)
  - $\Gamma \vdash e \Rightarrow \tau$ (e synthesizes τ)
- Introduction rules (constructors) are checked
- Elimination rules (destructors) synthesize

Bidirectional Type Checking

$$\frac{x \Rightarrow \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma}\ \text{hyp}$$

$$\frac{\Gamma, x \Rightarrow \sigma \vdash e \Leftarrow \tau}{\Gamma \vdash \lambda x.\, e \Leftarrow \sigma \to \tau}\ {\to}I \qquad \frac{\Gamma \vdash e_1 \Rightarrow \sigma \to \tau \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau}\ {\to}E$$

Bidirectional Type Checking

$$\frac{x \Rightarrow \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma}\ \text{hyp}$$

$$\frac{\Gamma, x \Rightarrow \sigma \vdash e \Leftarrow \tau}{\Gamma \vdash \lambda x.\, e \Leftarrow \sigma \to \tau} \to I \qquad \frac{\Gamma \vdash e_1 \Rightarrow \sigma \to \tau \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau} \to E$$

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' = \tau}{\Gamma \vdash e \Leftarrow \tau} \Rightarrow\Leftarrow$$

Bidirectional Type Checking

$$\frac{x \Rightarrow \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma}\ \text{hyp}$$

$$\frac{\Gamma, x \Rightarrow \sigma \vdash e \Leftarrow \tau}{\Gamma \vdash \lambda x.\, e \Leftarrow \sigma \to \tau}\ {\to}I \qquad \frac{\Gamma \vdash e_1 \Rightarrow \sigma \to \tau \quad \Gamma \vdash e_2 \Leftarrow \sigma}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau}\ {\to}E$$

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' = \tau}{\Gamma \vdash e \Leftarrow \tau}\ {\Rightarrow}{\Leftarrow}$$

- No type annotations in λ-abstractions
- With these rules, we can exactly type normal forms!

$$\begin{array}{llll} \text{Normal} & N & ::= & \lambda x.\, N \mid R \\ \text{Neutral} & R & ::= & x \mid R\, N \end{array}$$

Bidirectional Type Checking

- Add let form or type annotations

$$\frac{\Gamma \vdash e \Leftarrow \tau \quad \Gamma, x \Rightarrow \tau \vdash e' \Leftarrow \tau'}{\Gamma \vdash \textbf{let}\ x : \tau = e\ \textbf{in}\ e' \Leftarrow \tau'}\ \text{let}$$

- Properties
  - Concise (mostly annotating top level functions)
  - Increases compositionality by through stated types
  - Improves locality of error messages
  - Highly robust

Bidirectional Subtyping and Intersections

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' \leq \tau}{\Gamma \vdash e \Leftarrow \tau} \Rightarrow\Leftarrow$$

Bidirectional Subtyping and Intersections

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' \leq \tau}{\Gamma \vdash e \Leftarrow \tau} \Rightarrow\Leftarrow$$

$$\frac{\Gamma \vdash e \Leftarrow \tau \quad \Gamma \vdash e \Leftarrow \sigma}{\Gamma \vdash e \Leftarrow \tau \wedge \sigma} \wedge I$$

$$\frac{\Gamma \vdash e \Rightarrow \tau \wedge \sigma}{\Gamma \vdash e \Rightarrow \tau} \wedge E_1 \qquad \frac{\Gamma \vdash e \Rightarrow \tau \wedge \sigma}{\Gamma \vdash e \Rightarrow \sigma} \wedge E_2$$

Robustness

- How easily can type system features be extended or combined with other features?
- Example
  - Hindley-Milner type inference is extremely terse but relatively fragile
  - Pure type synthesis is verbose but robust
  - Bidirectional checking is concise and robust
- We don't know of another reasonable option for datasort refinements
- Bidirectional type checking is based on the logical notion of verification

Evaluation axis: How verbose are programs?

Evaluation axis: How robust are principles underlying the type system?

Lesson: Pay attention to usability in the software development and maintenance cycle

Lesson: Strive for using robust principles

Predictability

- Can we confidently predict if a program we write **should** type-check?
  - Predict yes, failure leads to debugging
  - Predict no, should reconsider or use dynamic techniques

Example: Standard Binary Numbers

- Binary numbers in standard form have no leading 0s

```
datatype bin = E | B0 of bin | B1 of bin
datasort std = E | B0 of pos | B1 of std
datasort pos =     B0 of pos | B1 of std

val zero : std
val succ : std -> pos
val pred : pos -> std
fun pred (B0(x)) = B1(pred x)
  | pred (B1(x)) = B0(x)
```

Example: Standard Binary Numbers

- Binary numbers in standard form have no leading 0s
- Type checking for `B0(x)` fails! $x \Rightarrow \mathit{std}$, but $\mathit{std} \not\leq \mathit{pos}$
- Indeed: `pred (B1(E)) = B0(E)` is not standard!

```
datatype bin = E | B0 of bin | B1 of bin
datasort std = E | B0 of pos | B1 of std
datasort pos =     B0 of pos | B1 of std

val zero : std
val succ : std -> pos
val pred : pos -> std
fun pred (B0(x)) = B1(pred x)
  | pred (B1(x)) = B0(x)
```

Predictability

- Data sorts can express exactly the properties of data types recognizable by finite tree automata
- Programs should check if the structure of the program follows the structure of sorts (which is often)
  - Sometimes we need to introduce additional sorts
  - Sometimes we need to ascribe additional sorts to have a fixed point
- Use dynamic coercions (partial and total) where information is not available

Sample Coercions

```
val std2pos : std -> pos  (* partial *)
fun std2pos E = error
  | std2pos (B0(x)) = B0(std2pos x)
  | std2pos (B1(x)) = B1(std2pos x)

val dbl : std -> std
fun dbl E = E
  | dbl x = B0(x)

val stdize : bin -> std   (* total *)
fun stdize E = E
  | stdize (B0(x)) = dbl (stdize x)
  | stdize (B1(x)) = B1 (stdize x)
```

Evaluation axis: How predictable is the type system?

Lesson: Type systems should be predictable, which comes from simplicity and uniformity

So Far . . .

- Simply-typed λ-calculus, ML
- Refinement types, including intersections
- Bidirectional type checking

So Far . . .

- Simply-typed λ-calculus, ML
- Refinement types, including intersections
- Bidirectional type checking
- Next: capturing intensional properties of programs

Runtime Code Generation

- Runtime code generation may improve efficiency, e.g.,
  - From standard to sparse matrix multiplication
  - From interpretation to compilation ($\sim$ partial evaluation)
- Language embeddings ($\sim$ macros)
- Problem: It often doesn't work, e.g.,
  - `mvmult : mat -> (vec -> vec)` could just build a closure
- Program must be properly staged

A Type for Quoted Expressions

- To compile at runtime we need source code
- Postulate a new type $\Box\tau$ for source expressions of type τ
- A function

  $$\mathsf{eval} : (\Box\alpha) \to \alpha$$

  compiles a quoted expression and then executes it
- Key idea: distinguish two kinds of variables
  - $x : \tau$, bound to values at runtime
  - $u : \tau$, bound to source expressions at runtime
- New expression context

  $$\Delta ::= \cdot \mid \Delta, u : \tau$$

Modal Typing

- Judgment $\Delta\,;\Gamma \vdash e : \tau$

$$\frac{\Delta\,;\cdot \vdash e : \tau}{\Delta\,;\Gamma \vdash \mathbf{box}\, e : \Box\tau}\;\Box I \qquad \frac{\Delta\,;\Gamma \vdash e : \Box\tau \quad \Delta, u : \tau\,;\Gamma \vdash e' : \tau'}{\Delta\,;\Gamma \vdash (\mathbf{let\,box}\, u = e \;\mathbf{in}\; e') : \tau'}\;\Box E$$

$$\frac{u : \tau \in \Delta}{\Delta\,;\Gamma \vdash u : \tau}\;\text{evar}$$

- A source expression cannot depend on value variables!
- Example

  eval : $\Box\alpha \to \alpha$

  eval $= \lambda x.\, \mathbf{let\,box}\; u = x \;\mathbf{in}\; u$

Example: Exponentiation

- Specify $\exp x\, b = b^x$
- Exploit $b^{2x} = (b * b)^x$, $b^{2x+1} = b * b^{2x}$
- Partial application just builds closure $\exp x = \lambda b. \ldots$

```
val exp : bin -> bin -> bin

fun exp E        b = B1(E)
  | exp (B0(x)) b = exp x (b * b)
  | exp (B1(x)) b = b * exp x (b * b)
```

Restage

```
val exp : bin -> [](bin -> bin)

fun exp E =         box (fn b => B1(E))
  | exp (B0(x)) = let box u = exp x
                  in box (fn b => u (b * b))
  | exp (B1(x)) = let box u = exp x
                  in box (fn b => b * u (b * b))
```

$$
\begin{aligned}
\mathsf{exp}\ 1 &= \mathbf{box}\ (\lambda b.\ b * (\lambda b'.\ 1)\ (b * b)) \\
&\simeq \mathbf{box}\ (\lambda b. b * 1) \\
\mathsf{exp}\ 2 &\simeq \mathbf{box}\ (\lambda b.(b * 1) * (b * 1)) \\
&\simeq \mathbf{box}\ (\lambda b. b * b)
\end{aligned}
$$

More Examples

- $\nvdash (\lambda x.\, \mathbf{box}\ x) : \alpha \to \Box\alpha$
- But, for every strictly positive type τ^+ we can define

$$\mathsf{lift}_{\tau^+} : \tau^+ \to \Box\tau^+$$

- Strictly positive types (but **not** functions or lazy pairs)

$$\tau^+ ::= 1 \mid \tau_1^+ \times \tau_2^+ \mid 0 \mid \tau_1^+ + \tau_2^+ \mid \mu\alpha^+.\, \tau^+ \mid \alpha^+$$

- Also

  $\mathsf{lift}_\Box : \Box\alpha \to \Box\Box\alpha$
  $\mathsf{lift}_\Box\ (\mathbf{box}\ u) = \mathbf{box}\ (\mathbf{box}\ u)$

Intuitionistic Modal Logic S4

- Axiomatically characterized by

$$\frac{\vdash A}{\vdash \Box A}\ \text{Nec}$$

$$\begin{array}{ll} \vdash \Box(A \to B) \to (\Box A \to \Box B) & \text{Normal} \\ \vdash \Box A \to A & \text{Reflexivity} \\ \vdash \Box A \to \Box\Box A & \text{Transitivity} \end{array}$$

- Coincides exactly with $\Box A$ for quoted expressions!
- Here: natural deduction rather than axiomatic proofs
- Quotation was one of the motivations for the development of modal logic in philosophy

Types as Propositions

- Example of the Curry-Howard correspondence
  - Propositions are types
  - Proofs are programs
  - Proof reduction is computation
- Helpful reference point in the design of type systems
  - Existence of a (good!) logic confirms validity of the abstraction
  - Helps with independence of operators from each other (robustness/modularity)
  - Metatheorems cleaner and simpler

More Generally . . .

- Co-develop type system and reasoning principles for programs in the language
  - A type system is not a goal in an of itself
  - The goal is to give the programmer the tools to express programs simply and correctly with the help of the type system
- We reason whenever we program!

  ```
  hd (sort A)
  ```

Lesson: Type systems are most effective if they reflect and validate the informal and intuitive reasoning that programmers do anyway

Lesson: Type systems are most effective if they reflect and validate the informal and intuitive reasoning that programmers do anyway or they introduce a new way to think about programs

Connecting Logic and Programming

logic	**computation**
intuitionistic logic	functional programming
modal logic S4	staged computation
modal logic S5	distributed computation
lax logic	monadic programming
discrete temporal logic	partial evaluation
singleton logic	linear communicating machines
linear logic	message-passing concurrency
temporal linear logic	timed concurrency

Connecting Proof Theory with PL Theory

proof theory	**programming language theory**
verifications	bidirectional type-checking
polarity	values vs. computations
polarity	sending vs. receiving
judgments vs. propositions	modes of computation
combinatory logic	combinatory reduction

Evaluation axis: Proximity between type system and logic

Lesson: Co-develop type system and reasoning principles

Lesson: Look for logical connections

Be relentless in your search for the simplest, most elegant abstractions that capture a phenomenon of interest

A Few Selected References

- Refinement types (data sorts)
  [Freeman & Pf, PLDI 1991] [Davies, AMAST 1997]
  [Dunfield & Pf, FoSSaCS 2003]
- Bidirectional type checking
  [Xi & Pf, POPL 1999] [Pf, ICFP 2007]
  [Dunfield & Krishnaswami, ICFP 2013]
- Staged computation
  [Davies & Pf, JACM 2001] [Pf & Davies, MSCS 2001]