

# Dependent types: Easy as PIE

(ongoing project report)

Dimitrios Vytiniotis and Stephanie Weirich  
University of Pennsylvania

TFP 2007, New York

# Programming with program properties

- Type systems goal: early error detection
  - ML family: ensure programs are crash-free
  - Modern view: types for **correctness properties**
    - Curry-Howard iso: types  $\leftrightarrow$  properties
    - Specifications written using types
    - Proofs  $\leftrightarrow$  programs

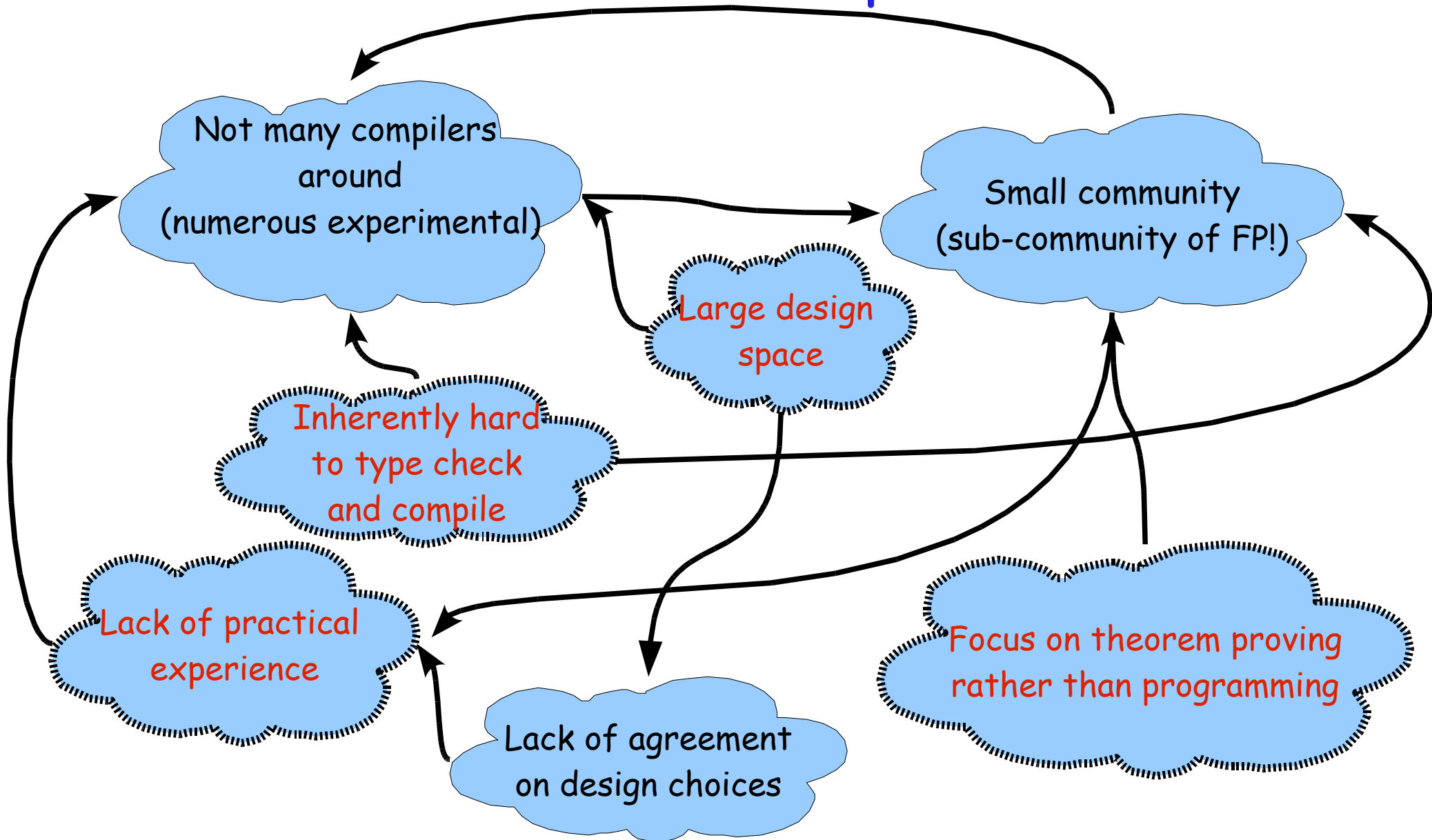
# Dependent types

- **THE** modern view on type systems:
  - precise types that depend on and describe values
  - can express elaborate program properties
  - can eliminate many runtime errors (and runtime checks!), such as pattern match failures
  - internal, parallel-to-development verification
- Coq, Epigram, Omega, Cayenne, ATS, RSP1, ...

But ...

... how many programmers actually use dependent types for day-to-day programming?

# The cause of the problem



# A new dependently typed language: PIE

- Explore new dimensions in **design space**
- Focus on **practical considerations**
  - Specify a simple and predictable system
  - Give programmers' needs high priority
  - Haskell-like familiar, functional, effectful

# Dependent types are already in use

- Think Haskell pseudo-dependently typed programming with *MPTCs/GADTs* etc.
- Uses separate language/programming paradigm for type/term level computations.
- Fascinating but scares new users away!
- ... what if we gave programmers an accessible way to program dependent types?

# PIE Fundamental approach: Terms as indices

- No duplication of program logic (**good**)
- No need to familiarize with two languages / styles of programming (**good**)
- Harder to separate “proofs” from computations (**bad**)
- Harder to enforce sound and decidable type equivalence checking (especially when effects are present!) (**bad**)

# Outline:

## Features we have decided on:

- Integration of effects and dependent types
- Semi-automatic exploitation of knowledge gained from pattern matching

## Future work (not in the talk):

- Phase distinction & static vs. dynamic terms
- Aggressive term & type inference

# Scope-safe de Bruijn-based AST

```
leq :: #Nat -> Nat -> Bool
```

```
leq Z _ = True
```

```
leq (S x) Z = False
```

```
leq (S x) (S y) = leq x y
```

```
data Eq :: Bool => Bool => * where
```

```
  Refl :: (b::Bool) -> Eq b b
```

```
data Term :: Nat => * where
```

```
  Var :: (m,n::Nat) -> Eq (leq (S m) n) True -> Term n
```

```
  Lam :: (n::Nat) -> Term (S n) -> Term n
```

```
  App :: (n::Nat) -> Term n -> Term n -> Term n
```

Totality check on argument:  
ensures leq is effect-free

(Effectful?) computation  
appears inside type:  
threatens sound and  
decidable type checking!

# PIE: Effect analysis for type indices

- Support for effects (incl. general recursion)
- But type-indexing expressions effect-free!
- Want to **rule out**:
  - $T(y:=0; \text{let } z = \text{ref } 1 \text{ in } !z) \stackrel{=?}{=} T(y:=1; 1)$
  - $T(\text{diverge}) \stackrel{=?}{=} T \text{ True}$
- ~~syntactic restrictions~~  $\rightarrow$  semantic
- Typing rules enforce indices be effect-free

# Propagation of effects

- Typing judgement extended with “static reduction effect”
  - Independent of runtime evaluation! (CBV/CBN)

$$G \vdash e : \tau @ \varphi$$

- **Type-level** applications well-formed when:

$$G \vdash \tau :: (x:s) \Rightarrow \kappa \qquad G \vdash e : s @ \downarrow$$

---

$$G \vdash \tau e :: \kappa\{e/x\}$$

# Treatment of variables in environments:

ALWAYS:  $G \vdash e : x @ \downarrow$

- lambda-bound  $x$ : “ $\lambda x \rightarrow \dots$ ” : no effect!
  - it cannot statically be reduced to anything!
- let-bound  $x$ , effect-free def.: “let  $x = e$  in ...”
  - can use definition during type equivalence!
- let-bound  $x$ , effectful def.: “let  $x = e$  in ...”
  - CANNOT reduce it, treat it **opaquely**!

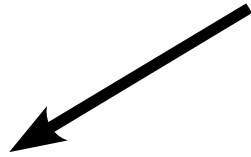
# Example: General recursion as effect (I)

- Type-based termination checking
- Based on work of Barthe et al. (LPAR06)
- Data types decorated with approximation indices ( $i$ ,  $i+$ ,  $\text{infinity}$ ):
  - Pattern matching reduces approximation index for sub-terms
- Subsumption induced by approximations:  
 $\text{Nat}(i) \prec: \text{Nat}(i+)$        $\text{Nat}(i) \prec: \text{Nat}(\text{inf})$

# General recursion as effect (II)

- Recursive definitions:

leq :: #Nat -> Nat -> Bool



leq: Nat(i) -> Nat(inf) -> Bool(inf) ⊢

\x y. ...leq body... : Nat(i+) -> Nat(inf) -> Bool(inf)

- “leq” can only be applied to “Nat(i)” arguments!

## General recursion as effect (III)

- If "leq" is applied to "x" ->> DO NOT REJECT
- ... but output an effect "↑" for the violation of the approximation index order:

$$\varphi := \uparrow \mid \downarrow$$

("↓" ~ non-effect)

# Pattern matching refinement

data Env :: Nat => \* where

Nil :: Env Z

Cons :: (n::Nat) -> Thunk -> Env n -> Env (S n)

$n = S\ n0, m = S\ m0$   
 $Eq\ (leq\ (S\ (S\ n0))\ (S\ m0))\ True$   
 $\leftrightarrow$   
 $Eq\ (leq\ (S\ n0)\ m0)\ True$

$Eq\ (leq\ (S\ n)\ m)$

lkup :: (n,m::Nat) -> Eq (leq (S n) m) True -> Env m -> Thunk

lkup Z \_ prf (Cons \_ thunk \_) = thunk

lkup (S n0) \_ **prf** (Cons m0 \_ hs) = lkup n0 m0 **prf** hs

No branch for "Nil"  
That would be inaccessible!

prf: only statically useful!

# The HOWs of pattern matching refinement

- Create **substitutions** expressing as many of the pattern match equalities as possible. E.g.

$$S = \{ n \rightarrow (S\ n0), m \rightarrow (S\ m0) \}$$

- Apply these substitutions when type checking the right-hand side of branches [even for variables in the environment, such as "prf" before! Feature absent from Coq!]
- UNLIKE Haskell+GADTs, not all constraints possible to express with substitutions!

# Hard constraints in pattern match

- Idea:

... case x of

J y1 ... yn **coerce** -> ....

- "coerce": Just an extra variable (like y1..yn)  
binds heterogeneous equality

coerce :: HEq x (J y1 ... yn)

- and "coerce" can now be used in the rhs!

# Route to adoption

## Haskell

- > [Add manually *SOME* dynamic checks—dependent types eliminate the need for many others]
- > [Integrate Verified and Non-Verified code (incremental verification)]
- > *PAY-AS-YOU-GO* approach!

# Future work

- Erasure of terms used statically. Phase distinction & proof irrelevance?
- Term/type inference + polymorphism
- Provable equality vs. convertibility
- Improve termination checking to mutual recursion/induction and coinductive types
- General effects and more precise analysis

Thank you

Questions & Discussion ...

# Provable equality vs. convertibility

- Convertibility: Full beta-reduction
- Provable equality: Reflexive equality encoded with a data type (Eq, or HEq)
- Convertibility is not enough:  
$$\text{Eq (leq n n) True} \not\leftrightarrow \text{Eq True True}$$
- Plan: (Heterogeneous) equality proof terms as ordinary terms. No proof scripts!
- Phase distinction to avoid runtime overhead!<sup>23</sup>

# The limits of static type checking

`f :: Nat -> Nat -> Nat -> Bool`

`f = ... encodes a predicate but produces effect! ...`

`bar :: (n,m,p::Nat) -> List n -> List m -> Eq (f n m p) True -> List p`

"f" will be treated abstractly (problematic)

- defer the check to runtime:

- hybrid type checking?

- ... or build a type level "F" predicate and a verifier!

# Computation indices vs. predicates

- E.g. RSP1: No computations inside types

```
data Leq :: Nat => Nat => * where
```

```
  Leq_base :: (n::Nat) -> Leq Z n
```

```
  Leq_ind  :: (n,m::Nat) -> Leq n m -> Leq (S n) (S m)
```

```
lkup :: (n, m :: Nat) -> Leq (S n) m -> Env m -> ...
```

- Unsatisfactory: Excessive user manipulation of proof objects, need for elaborate erasure procedure (lazyness helped here!)

# Recursive definitions and dynamic verification

- Must be treated **opaquely** if effect: ↑
- Important for post-design-time invariants!

`lkup :: (n::Nat) -> (env :: Env) -> (Eq (leq (S n) (length env)) True) -> ...`

`let env = ... potentially diverging ...`

`y = length env`

`in case (leq (S x) y) of`

`True prf -> lkup x env prf`

`Nothing -> ...`

extended pattern match  
construct  
`prf :: Eq (leq (S x) y) True`

`Eq (leq (S x) (length env)) True`  
  
env: EFFECTFUL!  
->> OK but treat it opaquely!

# Soundness of proof theory

- Soundness: No statically inaccessible branch may be accessed at runtime!
- CBV: Soundness modulo termination
- CBN: Proofs don't get evaluated:

`lkup :: (n,m::Nat) -> Eq (leq (S n) m) True -> Env m -> ...`

`... lkup (S Z) Z diverge Nil ...` **-->> BOOM!**

`bogus :: (n,m::Nat) -> Eq (leq (S n) m) True`

`bogus n m = case n of { }`

`... lkup (S Z) Z (bogus (S Z) Z) Nil` **-->> BOOM!**

# Precise effect analysis for soundness

- Proof use-site fix:

$\text{lkup} :: \dots \rightarrow \text{Terminating} (\text{Eq} (\text{leq} (\text{S } n) m) \text{True}) \rightarrow \dots$

- Precise effects:

$$G \vdash e : (\text{Int}@_{\varphi 1} \rightarrow \text{Int}@_{\varphi 2})@_{\varphi}$$

vs.

$$G \vdash e : (\text{Int} \rightarrow \text{Int})@_{\varphi}$$

- + coverage checking!

# A simple idea for coverage checking

- Allow programmers to explicitly declare when they believe branches are inaccessible:

- $p :: (n,m::\text{Nat}) \rightarrow \text{Eq} (\text{leq} (S\ n)\ m)\ \text{True} \rightarrow \text{Env}\ m \rightarrow \{p::\text{Nat}, \text{Term}\ p, \text{Env}\ p\}$

$\text{lkup}\ Z\ \_ \text{prf}\ \text{env} = \text{case}\ \text{env}\ \text{of}$

$\text{Nil} \rightarrow \text{inaccessible}$

$\text{Cons}\ \_ \text{hind}\ h\ \text{henv}\ \_ \rightarrow \{\text{hind}, h, \text{env}\}$

$\text{lkup}\ (S\ n0)\ \_ \text{prf}\ \text{env} = \dots \text{similarly} \dots$

- Automatically expand nested patterns