## A CORE LANGUAGE FOR DEPENDENTLY TYPED PROGRAMMING THORSTEN ALTENKIRCH NICOLAS OURY UNIVERSITY OF NOTTINGHAM

## MOTIVATIONS

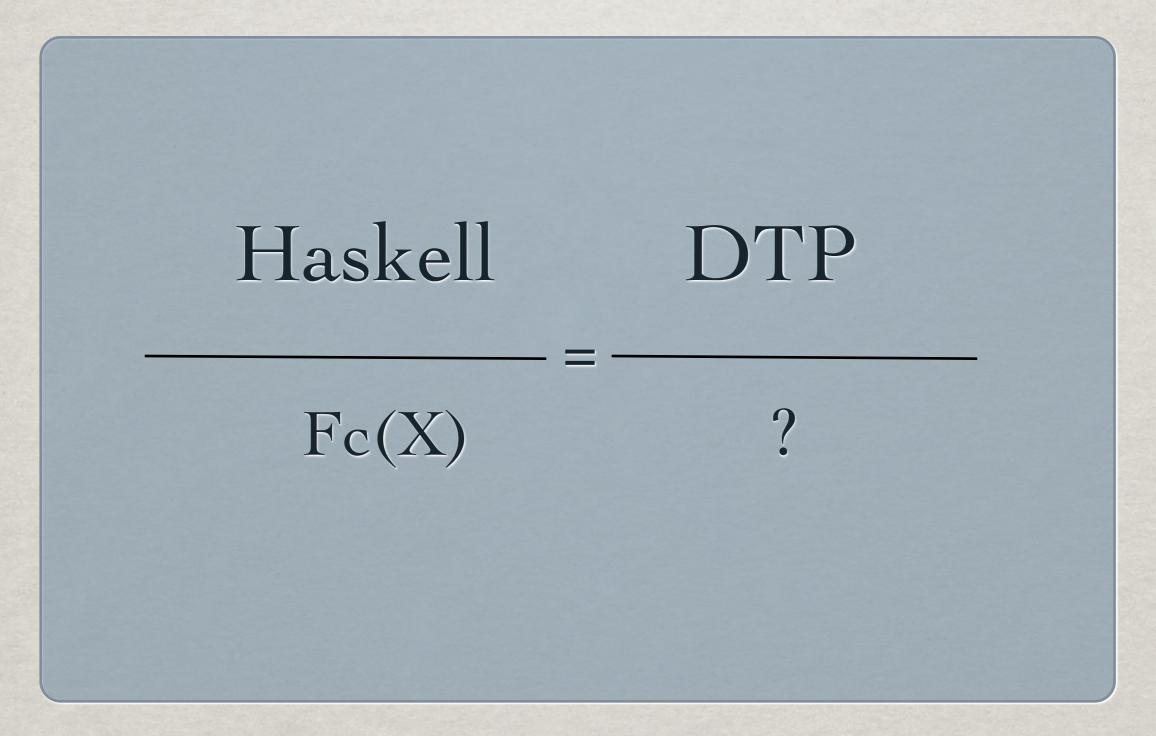
Dependently typed languages

(for programs and proofs)

e.g. CIC (Coq), Epigram, Agda, Cayenne ...

- Factor implementation into core language and high level language.
- Core language should be independent of your notion of totality.







- Small and simple
  - Meta-theory feasible
- Batch compilation
  - No interactive development necessary
- Yet sufficiently general

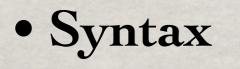
## DESIGN IDEAS

## **GENERAL RECURSION**

- Allow mutual recursive definitions
- Typing assumptions and recursive definitions may depend on each other.
- Syntax

## **GENERAL RECURSION**

- Allow mutual recursive definitions
- Typing assumptions and recursive definitions may depend on each other.



### UNIVERSES

- General recursion makes the system logically inconsistent
- So we don't lose anything by having

Type : Type

• This allows to simulate any universes hierarchy

## FINITE TYPES

• Set of labels is a type:

• Typing a label:

• Case analysis:  $t: \{A, B, C\}$  $case t of \{ A \rightarrow ... \\ B \rightarrow ... \\ C \rightarrow ... \}$ 

## **Π-Types**

- Nothing really new here
- **П**-types :

$$(x:A) \rightarrow B[x]$$

• Inhabited by functions:

$$\setminus x \rightarrow t [x]$$

• Eliminated by application:



• A type for dependent pair:

x : A; B [x]

(u, v)

• Introduce by pairing:

• Elimination by a **letp** operator:

$$letp (x,y) = p in t$$

### FEATURES SUMMARY

- General recursion
- Very impredicative universe
- Finite type,  $\Pi$ -Types,  $\Sigma$ -Types
- We postpone equality types
- That's all: simple but sufficient

# ENCODING COMPLEX TYPES

## **ENCODING TYPES**

#### • Labeled sums:

Either : Type  $\rightarrow$  Type  $\rightarrow$  Type Either =  $\land A \land B \rightarrow$  tag : {Left, Right}; case tag of {Left  $\rightarrow A \mid \text{Right} \rightarrow B$ }

#### • Recursive types:

Nat : Type  
Nat = tag : 
$$\{Z, S\}$$
 ; case tag of  $\{Z \rightarrow \{Void\} | S \rightarrow Nat \}$ 

## **ENCODING TYPES**

#### • Labeled sums:

Either : Type  $\rightarrow$  Type  $\rightarrow$  Type Either =  $\land A \land B \rightarrow$  tag : {Left, Right}; case tag of {Left  $\rightarrow A \mid \text{Right} \rightarrow B$ }

• Recursive types:

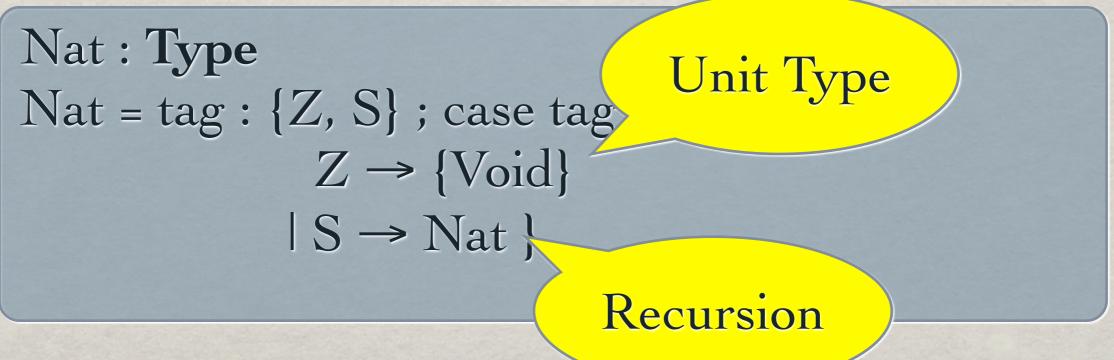
Nat : Type  
Nat = tag : {Z, S} ; case tag  
$$Z \rightarrow \{Void\}$$
  
 $|S \rightarrow Nat \}$ 

## **ENCODING TYPES**

#### • Labeled sums:

Either : Type  $\rightarrow$  Type  $\rightarrow$  Type Either =  $\land A \land B \rightarrow$  tag : {Left, Right}; case tag of {Left  $\rightarrow A \mid \text{Right} \rightarrow B$ }

• Recursive types:



## FAMILIES OF TYPES

Vec : Type 
$$\rightarrow$$
 Nat  $\rightarrow$  Type  
Vec =  $\setminus$  A n  $\rightarrow$  letp (tag, n') = n in  
case tag of {  
 $Z \rightarrow l:{Nil}; Void$   
 $\mid S \rightarrow l:{Cons}; A; Vec A n'}$ 

FAMILIES OF  

$$Vec : Type \rightarrow Nat \rightarrow Type$$
  
 $Vec = \land A n \rightarrow letp (tag, n') = n in$   
 $case tag of \{$   
 $Z \rightarrow l: \{Nil\}; Void$   
 $\mid S \rightarrow l: \{Cons\}; A; Vec A n'\}$ 

FAMILIES OF  

$$Vec : Type \rightarrow Nat \rightarrow Type$$
  
 $Vec = \setminus A n \rightarrow letp (tag, n') = n in$   
 $case tag of \{$   
 $Z \rightarrow l: \{Nil\}; Void$   
 $\mid S \rightarrow l: \{Cons\}; A; Vec A n'\}$ 

Fin : Nat 
$$\rightarrow$$
 Type  
Fin = \n  $\rightarrow$  letp (tag, n') = n in  
case tag of {  $Z \rightarrow$  {}|S  $\rightarrow$  1 : { $Z$ , S};  
case l of { $Z \rightarrow$  {Void}  
S  $\rightarrow$  Fin n'}}

## DIY EQUALITY

• Encoding equality of natural numbers:

 $Eq: Nat \rightarrow Nat \rightarrow Type$  $Eq = \langle n m \rightarrow letp (ln, n') = n in$ **letp** (lm, m') = m **in** case ln of {  $Z \rightarrow case lm of \{ Z \rightarrow \{Void\} \mid S \rightarrow \{ \} \}$  $| S \rightarrow case lm of \{$  $Z \rightarrow \{\}$  $| S \rightarrow Eq n'm' \}$ 

## **A UNIVERSE**

U : Type  
El : U 
$$\rightarrow$$
 Type  
U = l: {u, pi} ; case l of {  
 $u \rightarrow$  {Void}  
 $pi \rightarrow a : U; El a \rightarrow U$ }  
El = \a  $\rightarrow$  letp (l;node) = a in case l of {  
 $u \rightarrow A$   
 $pi \rightarrow$  letp (src, tgt) = node in  
 $(x : El src) \rightarrow El (tgt x)$ 

## MAIN ISSUES



• Looping with general recursion

• Pattern matching

### LOOPING

- General recursion makes type checking undecidable
- Type checker may loop because a term doesn't terminate
- Requirement: type checker should not loop for *reasonable* programs.

## LOOPING: IDEA

• We sometimes put a **box** around a part of the context:

 $\Gamma, [\Gamma'], \Gamma'' \vdash t : T$ 

• A recursive definition can only be used when **not** in a **box** 

$$\dots, f \to u, \dots \vdash f \equiv u$$

## **BOXES: WHEN?**

- We want to prevent looping of a definition fact =  $\ n \rightarrow \dots$  case tag of  $Z \rightarrow fact n' \dots$
- We need to box recursive calls of a function
- We do this by putting a box on the context when we meet a **case**

 $[\Gamma] \vdash b_i : T$ 

 $\Gamma \vdash \mathsf{case} \ e \ \mathsf{of} \ \{L_i \to b_i, \ldots\} : T$ 

## **BOXES: WHEN?**

unfolds to:

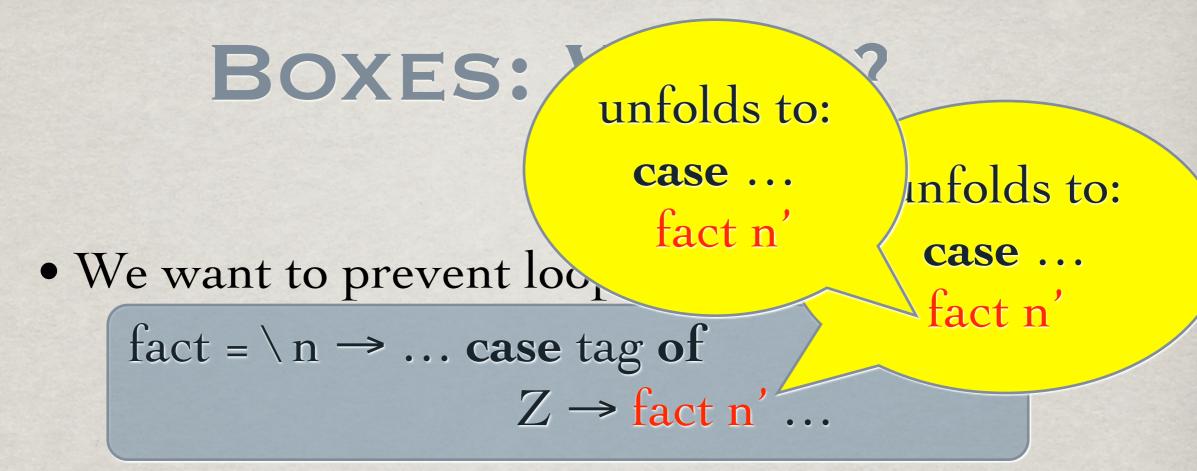
**case** ...

fact n'

- We want to prevent looping of a fact =  $\ n \rightarrow \dots$  case tag of  $Z \rightarrow fact n' \dots$
- We need to box recursive calls of a function
- We do this by putting a box on the context when we meet a **case**

 $[\Gamma] \vdash b_i : T$ 

 $\Gamma \vdash \mathsf{case} \ e \ \mathsf{of} \ \{L_i \to b_i, \ldots\} : T$ 



- We need to box recursive calls of a function
- We do this by putting a box on the context when we meet a **case**

 $[\Gamma] \vdash b_i : T$ 

$$\Gamma \vdash \mathsf{case} \ e \ \mathsf{of} \ \{L_i \to b_i, \ldots\} : T$$

## BOXES AND COMPUTATIONS

• We need to do some computations

• What happens here?

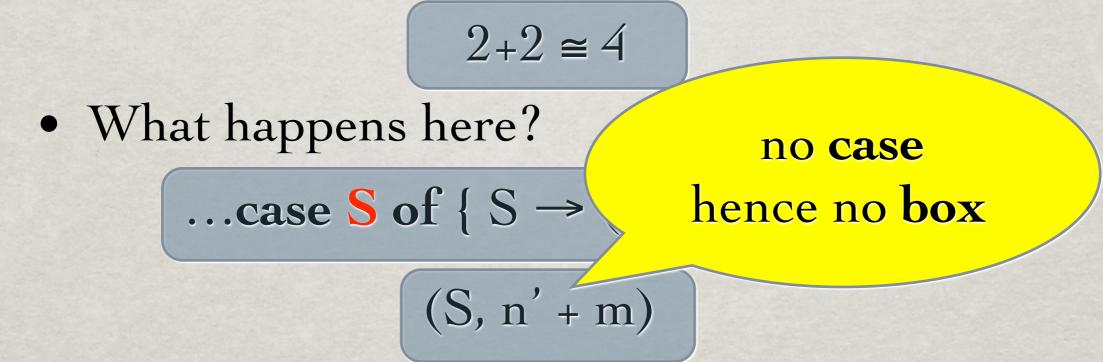
...case **S** of {  $S \rightarrow (S, n' + m) \dots$ 

(S, n' + m)

• Reduction occurs when there is no stuck elimination

## BOXES AND COMPUTATIONS

• We need to do some computations



• Reduction occurs when there is no stuck elimination

## PATTERN MATCHING

- Agda: Pattern matching primitive
- Epigram: Generating *motives* for standard eliminators.
- Coq: Under discussion
- Our proposal: use of constraints Advantages: local case (with) is easy less complexity in the translation

### EXAMPLE

append :: (n m)  $\rightarrow$  Vect n  $\rightarrow$  Vect m  $\rightarrow$  Vect (n + m) append = \ n m xs ys  $\rightarrow$  letp (tagn, n') = n (tagxs, xs') = xs in case tagn of {  $Z \rightarrow$  case tagxs of { Nil  $\rightarrow$  ys }

> S → case tagxs of { Cons→ (Cons, append n' m xs' ys)}

#### EXAMPLE

append :: (n m)  $\rightarrow$  Vect n  $\rightarrow$  Vect m  $\rightarrow$  Vect (n + m) append = \ n m xs ys  $\rightarrow$  letp (tagn, p') n (tr tagn=Z case tagn of {  $Z \rightarrow$  case tagxs of {  $Nil \rightarrow ys$ }

> S → case tagxs of { Cons→ (Cons, append n' m xs' ys)}

### EXAMPLE

append :: (n m)  $\rightarrow$  Vect n  $\rightarrow$  Vect m  $\rightarrow$  Vect (n + m) append = \ n m xs ys  $\rightarrow$  letp (tagn, p') n (t' tagn=Z case tagn of {  $Z \rightarrow$  case tagxs of { Nil  $\rightarrow$  ys } n = (S,n') n+m = (S,n'+m)

> $S \rightarrow case tagxs of {$  $Cons \rightarrow (Cons, append n'm xs'ys)}$

### CONSTRAINTS

• Case analysis for simple types:

 $\frac{\Gamma \vdash e : \{l_1, \dots, l_n\} \qquad \Gamma \vdash t_i : T}{\Gamma \vdash \mathsf{case} \ e \ \mathsf{of}\{\dots | l_i \to t_i | \dots\} : T}$ 

• Case analysis with constraints:

$$\frac{\Gamma \vdash e : \{l_1, \dots, l_n\}}{\Gamma \vdash \mathsf{case} \ e \ \mathsf{of}\{\dots | l_i \to t_i | \dots\} : T}$$

### EXAMPLES

So : {True, False}  $\rightarrow$  Type So =  $\b \rightarrow$  case b of {True  $\rightarrow$  {Void} | False  $\rightarrow$  {}} reflNat : (n:Nat)  $\rightarrow$  So (eq n n). reflNat =  $\ n \rightarrow$ letp(nl,n') = n incase nl of {  $Z \rightarrow Void$  $| S \rightarrow reflNat n' \}$ 

So : {True, False}  $\rightarrow$  Type So =  $\b \rightarrow$  case b of {True  $\rightarrow$  {Void} | False  $\rightarrow$  {}} reflNat : (n:Nat)  $\rightarrow$  So (eq n n) nl≡Z reflNat =  $\ n \rightarrow$ SO letp(nl,n') = n in $eq n n \equiv {Void}$ case nl of {  $Z \rightarrow Void$  $| S \rightarrow reflNat n' \}$ 

So : {True, False}  $\rightarrow$  Type So =  $\b \rightarrow$  case b of {True  $\rightarrow$  {Void} | False  $\rightarrow$  {}} reflNat : (n:Nat)  $\rightarrow$  So (eq n n) nl≡Z reflNat =  $\ n \rightarrow$ SO letp(nl,n') = n in $eq n n \equiv {Void}$ case nl of {  $Z \rightarrow Void$  $| S \rightarrow reflNat n' \rangle$ nl≡S SO eq n n ≡ eq n' n'

filter : (A)  $\rightarrow$  (A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  List A. filter = ... all : (p : A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  Bool all = ...

 $prop : (A p) \rightarrow (as:List A) \rightarrow So (all A p (filter A p as))$   $prop = \langle A p as \rightarrow letp (tag, node) = as in$   $case tag of \{$   $Nil \rightarrow Void$   $Cons \rightarrow letp (a, as') = node in$   $case p a of \{$   $True \rightarrow prop A p as'$   $False \rightarrow prop A p as' \}\}$ 

filter : (A)  $\rightarrow$  (A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  List A. filter = ... all : (p : A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  Bool all = ...

prop : (A p)  $\rightarrow$  (as:List A)  $\rightarrow$  So (all A p (filter A p as)) prop = \ A p as  $\rightarrow$  letp case tag of { Nil  $\rightarrow$  Void Cons  $\rightarrow$  letp (a,as') = node in case p a of { True  $\rightarrow$  prop A p as' False  $\rightarrow$  prop A p as' }}

filter : (A)  $\rightarrow$  (A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  List A. filter = ... all : (p : A  $\rightarrow$  Bool)  $\rightarrow$  List A  $\rightarrow$  Bool all = ...

prop : (A p)  $\rightarrow$  (as:List A)  $\rightarrow$  So (all A p (filter A p as)) prop = \A p as  $\rightarrow$  letp case tag of { Nil  $\rightarrow$  Void Cons  $\rightarrow$  letp (a,a) case p a of { True  $\rightarrow$  prop A p as' False  $\rightarrow$  prop A p as' }}

# PROTOTYPE

## PROTOTYPE

- Some design choices:
  - Bidirectional type checking
  - Typed equality test
- Constraints:
  - rewrite rules applied to head of values
  - naive but works on examples

## PROTOTYPE

- Implementing general recursion Can be difficult to restart evaluation when unfolding a definition.
- We glue together a neutral with its content

• We use laziness to postpone evaluation of v

# FUTURE WORK

# **GENERAL CONSTRAINTS**

• Add any constraint to the type checker Type "T if u and v are convertible"

$$\{u \equiv v\} \Rightarrow T$$

Type "T and I ensure that u and v are convertible"  $\{T \mid u \equiv v\}$ 

• Encode equality type

 $eq u v = \{\{Void\} \mid u \equiv v\}$ 

# **GENERAL CONSTRAINTS**

- What kind of constraints? It may be possible to include constraints between **constructors**, **tuples** and **neutral terms**.
- In a given context, all these are order 0 terms.

• For higher order, use an Observational Type Theory like equality.

# **GENERAL BOXES**

- We protect recursion under cases
- We can add user specified **boxes** Specify not to unfold recursion in [t]
- Example: co-data

stream :  $(A : Type) \rightarrow Type$ stream =  $\setminus A \rightarrow l: \{Cons\}; A;$ case l of { Cons  $\rightarrow$  (stream A)-} zeros : stream Nat zeros = 0, [ zeros ]

[t] : T-

## GENERAL BOXES

- To compute we need to open a box
   open [t] = t
- Our boxes are a special case :
   open (case e of { ... → [t]})
- Working with codata

tail : stream A  $\rightarrow$  stream A tail = \ xs  $\rightarrow$  letp (tag, node) = xs in case tag of {Cons  $\rightarrow$  letp (\_, tl) = node in open tl }

# MORE TO DO

- Integrate meta-variables. May have strange interaction with constraints.
- Reflection and generic programming.
- Phase separation and compiler.
- Evidence based optimization.