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.

Equation

$$\frac{\text{Haskell}}{\text{Fc(X)}} = \frac{\text{DTP}}{?}$$

Goals

- 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**

```
let { x : U
      x = u [x]
      y : V [x]
      y = v [x, y] } in t[x, y]
```

General Recursion

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

```
let { x : U
      x = u [x]
      y : V [x]
      y = v [x, y] } in t[x, y]
```

depends on x = u [x]

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: $\{A,B,...\} : Type$
- Typing a label: $L : \{..., L, ...\}$
- Case analysis:

```
case t of {
      A → ...
    | B → ...
    | C → ...}
```

t :{A, B, C}

Π-Types

- Nothing really new here

- Π-types :

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

- Inhabited by functions: $\backslash x \to t\ [x]$

- Eliminated by application: $f\ \ t$

Σ-Types

- A type for dependent pair:

  x : A; B [x]

- Introduce by pairing: (u, v)
- Elimination by a **letp** operator:

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

Features Summary

- General recursion
- Very impredicative universe
- Finite type, Π-Types, Σ-Types
- We postpone equality types
- That's all: simple but sufficient

Encoding Complex Types

Encoding Types

- Labeled sums:

```
Either : Type → Type → Type
Either = \ A B → tag : {Left, Right};
    case tag of {Left →A  | Right → B}
```

- Recursive types:

```
Nat : Type
Nat = tag : {Z, S} ; case tag of {
                Z → {Void}
               | S → Nat }
```

Encoding Types

- Labeled sums:

```
Either : Type → Type → Type
Either = \ A B → tag : {Left, Right};
    case tag of {Left →A | Right → B}
```

- Recursive types:

```
Nat : Type
Nat = tag : {Z, S} ; case tag
              Z → {Void}
             | S → Nat }
```

Unit Type

Encoding Types

- Labeled sums:

```
Either : Type → Type → Type
Either = \ A B → tag : {Left, Right};
    case tag of {Left →A | Right → B}
```

- Recursive types:

```
Nat : Type
Nat = tag : {Z, S} ; case tag
            Z → {Void}
          | S → Nat }
```

Unit Type

Recursion

Families of types

```
Vec : Type → Nat → Type
Vec = \ A n → letp (tag, n') = n in
    case tag of {
        Z → l:{Nil}; Void
      | S → l:{Cons}; A; Vec A n'}
```

Families of

Remember
Nat is a pair

```
Vec : Type → Nat → Type
Vec = \ A n → letp (tag, n') = n in
    case tag of {
        Z → l:{Nil}; Void
      | S → l:{Cons}; A; Vec A n'}
```

Families of

Remember Nat is a pair

```
Vec : Type → Nat → Type
Vec = \ A n → letp (tag, n') = n in
    case tag of {
        Z → l:{Nil}; Void
      | S  → l:{Cons}; A; Vec A n'}
```

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

DIY Equality

- Encoding equality of natural numbers:

```
Eq : Nat → Nat → Type
Eq = \ n m → letp (ln, n')    = n in
              letp (lm, m') = m in
  case ln of {
      Z → case lm of { Z → {Void} | S → { }}
   |  S  → case lm of {
              Z → { }
            | S → Eq n' m' }
```

A Universe

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

Main Issues

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**

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

Boxes: When?

- We want to prevent looping of a definition

```
fact = \ n → … case tag of
                    Z → fact n' …
```

- We need to **box** recursive calls of a function

- We do this by putting a box on the context when we meet a **case**

$$\frac{[\Gamma] \vdash b_i : T \qquad \ldots}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{L_i \rightarrow b_i, \ldots\} : T}$$

Boxes: When?

- We want to prevent looping of a

```
fact = \ n → … case tag of
                     Z → fact n' …
```

unfolds to:
case …
fact n'

- We need to **box** recursive calls of a function

- We do this by putting a box on the context when we meet a **case**

$$\frac{[\Gamma] \vdash b_i : T \qquad \ldots}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{L_i \to b_i, \ldots\} : T}$$

Boxes: ?

unfolds to:
case …
fact n’

unfolds to:
case …
fact n’

- We want to prevent loo

```
fact = \ n → … case tag of
                 Z → fact n’ …
```

- We need to **box** recursive calls of a function

- We do this by putting a box on the context when we meet a **case**

$$\frac{[\Gamma] \vdash b_i : T \qquad \ldots}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{L_i \to b_i, \ldots\} : T}$$

Boxes and Computations

- We need to do some computations

$$2+2 \cong 4$$

- What happens here?

…**case** S **of** { S → (S, n' + m) …

(S, n' + m)

- Reduction occurs when there is no stuck elimination

Boxes and Computations

- We need to do some computations

$2+2 \cong 4$

- What happens here?

…**case** S **of** { S → (

(S, n' + m)

no **case**
hence no **box**

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


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

Example

```
append :: (n m) → Vect n → Vect m → Vect (n + m)
append = \ n m xs ys → letp (tagn, n’)    n
                              (t
    case tagn of {
          Z → case tagxs of {
                    Nil    → ys }


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

tagn≡Z

so

n+m ≡ m

Example

```
append :: (n m) → Vect n → Vect m → Vect (n + m)
append = \ n m xs ys → letp (tagn, n’) = n
                        (t
    case tagn of {
        Z → case tagxs of {
                Nil    → ys }


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

tagn≡Z
so
n+m ≡ m

n ≡ (S,n’)
n+m ≡ (S,n’+m)

Constraints

- Case analysis for simple types:

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

- Case analysis with constraints:

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

Examples

```
So : {True, False} → Type
So = \ b →  case b of {True → {Void} | False → {}}

reflNat : (n:Nat) → So (eq n n).
reflNat = \ n  →
        letp (nl,n') = n in
              case nl of {
                 Z → Void
               | S → reflNat n' }
```

Examples

```
So : {True, False} → Type
So = \ b →  case b of {True → {Void} | False → {}}

reflNat : (n:Nat) → So (eq n n).
reflNat = \ n  →
          letp (nl,n') = n in
                case nl of {
                   Z → Void
                 | S → reflNat n' }
```

nl≡Z
so
eq n n ≡ {Void}

Examples

```
So : {True, False} → Type
So = \ b →  case b of {True → {Void} | False → {}}

reflNat : (n:Nat) → So (eq n n).
reflNat = \ n  →
        letp (nl,n') = n in
              case nl of {
                 Z → Void
                | S → reflNat n' }
```

nl≡Z
so
eq n n ≡ {Void}

nl≡S
so
eq n n ≡ eq n' n'

Examples

```
filter : (A) → (A → Bool) → List A → List A.
filter = …
all : (p : A → Bool) → List A → Bool
all = …

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

Examples

```
filter : (A) → (A → Bool) → List A → List A.
filter = …
all : (p : A → Bool) → List A → Bool
all = …

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

So True

Examples

filter : (A) → (A → Bool) → List A → List A.
filter = …
all : (p : A → Bool) → List A → Bool
all = …

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

So True

p a ≡ True so
all (filter a:as') ≡ all (filter 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

x t … [:= v]

- 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]

[t] : T-

- Example: **co-data**

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

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 → stream A
tail = \ xs → letp (tag, node) = xs in
     case tag  of
        {Cons → 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.