A Core Language
For DTP

T. AlteneKirch
N. Oury
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.
Haskell = DTP

\[ Fc(X) \]
Goals

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

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

```plaintext
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

\[ \text{let } \{ \begin{align*} x &: U \\
&= u [x] \\
y &: V [x] \\
y &= v [x, y] \end{align*} \} \text{ in } t[x, y] \]

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 \rightarrow ...
  | B \rightarrow ...
  | C \rightarrow ...
}
t :{A, B, C}
• Nothing really new here

• Π-types:

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

• Inhabited by functions:

\[\lambda x \rightarrow t [x]\]

• Eliminated by application:

\[f \quad t\]
**Σ-TYPES**

- A type for dependent pair:
  $$x : A; B \ [x]$$

- Introduce by pairing:
  $$(u, v)$$

- Elimination by a `letp` operator:
  $$\text{letp} \ (x, y) = p \ \text{in} \ t$$
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 → 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 = \( A \times B \rightarrow \text{tag} : \{\text{Left, Right}\}; \)
\[ \text{case tag of } \{\text{Left} \rightarrow A | \text{Right} \rightarrow B\} \]

• Recursive types:

Nat : Type
Nat = \( \text{tag} : \{Z, S\} ; \text{case tag} \)
\[ Z \rightarrow \{\text{Void}\} \]
\[ | S \rightarrow \text{Nat} \}

Unit Type
Encoding Types

- Labeled sums:

$$\text{Either} : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$$

$$\text{Either} = \lambda A \ B \rightarrow \text{tag} : \{\text{Left, Right}\};$$

$$\text{case tag of } \{\text{Left} \rightarrow A \mid \text{Right} \rightarrow B\}$$

- Recursive types:

$$\text{Nat} : \text{Type}$$

$$\text{Nat} = \text{tag} : \{\text{Z, S}\}; \text{case tag of } \{\text{Z} \rightarrow \{\text{Void}\} \mid \text{S} \rightarrow \text{Nat}\}$$
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 Types

Vec : Type → Nat → Type

Vec = \( A \; n \rightarrow \text{letp} \; (\text{tag}, \; n') = n \; \text{in} \)

\[
\text{case tag of }\
\quad Z \rightarrow l:\{\text{Nil}\}; \; \text{Void} \\
\quad S \rightarrow l:\{\text{Cons}\}; \; A; \; \text{Vec} \; A \; n'
\]

Remember Nat is a pair
Families of Types

Vec : Type → Nat → Type
Vec = λA n → letp (tag, n’) = n in
  case tag of {
    Z → l:[Nil]; Void
    | S → l:[Cons]; A; Vec A n’

Fin : Nat → Type
Fin = λn → 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:

\[
\text{Eq} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Type} \\
\text{Eq} = \lambda n \ m \rightarrow \text{letp} \ (ln, n') = n \ \text{in} \\
\quad \text{letp} \ (lm, m') = m \ \text{in} \\
\quad \text{case} \ ln \ of \ \{ \\
\quad \quad Z \rightarrow \text{case} \ lm \ of \ \{ \\
\quad \quad \quad Z \rightarrow \{\text{Void}\} \mid S \rightarrow \{\} \} \\
\quad \mid S \rightarrow \text{case} \ lm \ of \ \{ \\
\quad \quad Z \rightarrow \{\} \\
\quad \mid S \rightarrow \text{Eq} \ n' \ m' \} \\
\}
\]
A Universe

\[
\begin{align*}
U &: \text{Type} \\
El &: U \rightarrow \text{Type} \\
U &= l:\{u, \pi\} ; \text{case } l \text{ of } \{ \\
&\quad u \rightarrow \{\text{Void}\} \\
&\quad \pi \rightarrow a : U; \text{El } a \rightarrow U\} \\
El &= \lambda a \rightarrow \text{let } p (l;\text{node}) = a \text{ in case } l \text{ of } \{ \\
&\quad u \rightarrow A \\
&\quad \pi \rightarrow \text{let } p (\text{src, tgt}) = \text{node } \text{ in } \\
&\qquad (x : \text{El } \text{src}) \rightarrow \text{El } (\text{tgt } x)\}
\end{align*}
\]
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.
Loopy: 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

\[\ldots, f \rightarrow u, \ldots \vdash f \equiv u\]
We want to prevent looping of a definition

\[
\text{fact} = \lambda n \rightarrow \ldots \text{case tag of} \\
Z \rightarrow \text{fact } n' \ldots
\]

We need to box recursive calls of a function

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

\[
[\Gamma] \vdash b_i : T \\
\ldots
\]

\[
\Gamma \vdash \text{case } e \text{ of } \{ L_i \rightarrow b_i, \ldots \} : T
\]
Boxes: When?

• We want to prevent looping of a recursive definition

\[
\text{fact} = \lambda n \rightarrow \ldots \text{case tag of } Z \rightarrow \text{fact } n' \ldots
\]

• We need to box recursive calls of a function

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

\[
[\Gamma] \vdash b_i : T \quad \ldots
\]

\[
\Gamma \vdash \text{case } e \text{ of } \{ L_i \rightarrow b_i, \ldots \} : T
\]
Boxes: Why?

- We want to prevent looping of a definition

\[
\text{fact} = \lambda n \rightarrow \ldots \text{case tag of} \\
Z \rightarrow \text{fact } n' \ldots
\]

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

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

\[
[\Gamma] \vdash b_i : T \quad \ldots \\
\hline
\Gamma \vdash \text{case } e \text{ of } \{ L_i \rightarrow b_i, \ldots \} : T
\]
• We need to do some computations

\[ 2 + 2 \cong 4 \]

• What happens here?

\[ \text{case } S \text{ of } \{ S \rightarrow (S, n' + m) \ldots \] 

\[ (S, n' + m) \]

• Reduction occurs when there is no stuck elimination
• We need to do some computations

\[ 2 + 2 \equiv 4 \]

• What happens here?

... case S of \{ S \rightarrow \}

(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

\[
\text{append} :: (n \ m) \to \text{Vect} \ n \to \text{Vect} \ m \to \text{Vect} \ (n + m)
\]

\[
\text{append} = \lambda n \ m \ \text{xs} \ \text{ys} \to \text{let} \ \begin{align*}
& (\text{tagn}, \ n') = n \\
& (\text{tagxs}, \ \text{xs}') = \text{xs} \ \text{in} \\
& \text{case} \ \text{tagn} \ \text{of} \ \{ \\
& \quad \text{Z} \to \text{case} \ \text{tagxs} \ \text{of} \ \{ \\
& \quad \quad \text{Nil} \to \text{ys} \\
& \quad \}\\\
& \quad \text{S} \to \text{case} \ \text{tagxs} \ \text{of} \ \{ \\
& \quad \quad \text{Cons} \to (\text{Cons}, \ \text{append} \ n' \ m \ \text{xs}' \ \text{ys}) \\
& \}\end{align*}
\]
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)}

  tagn ≡ Z
  so
  n+m ≡ m

S → case tagxs of {
  Cons → (Cons, append n' m xs' ys)}
append :: (n m) → Vect n → Vect m → Vect (n + m)
append = \n m xs ys → letp (tagn, n') = n
(tagxs, xs') = xs

   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:

\[
\Gamma \vdash e : \{ l_1, \ldots, l_n \} \quad \Gamma \vdash t_i : T \\
\Gamma \vdash \text{case } e \text{ of } \{ \ldots | l_i \to t_i | \ldots \} : T
\]

- Case analysis with constraints:

\[
\Gamma \vdash e : \{ l_1, \ldots, l_n \} \quad \Gamma, e \equiv l_i \vdash t_i : T \\
\Gamma \vdash \text{case } e \text{ of } \{ \ldots | l_i \to t_i | \ldots \} : T
\]
Examples

So : \{True, False\} \rightarrow Type
So = \ b \rightarrow \ case \ b \ of \ \{True \rightarrow \{Void\} \mid False \rightarrow {}\}\)

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

So : \{True, False\} \rightarrow Type
So = \(\lambda b \rightarrow \text{case } b \text{ of } \{\text{True } \rightarrow \{\text{Void}\} \mid \text{False } \rightarrow \{\}\}\)
**Examples**

So : \{True, False\} \rightarrow Type
So = \lambda b \rightarrow \text{case } b \text{ of } \{\text{True } \rightarrow \{\text{Void}\} \mid \text{False } \rightarrow \{}\}

reflNat : (n:Nat) \rightarrow So (eq n n).
reflNat = \lambda n \rightarrow
  \text{letp } (nl,n') = n \text{ in }
  \text{case } nl \text{ of } \{
    Z \rightarrow \text{Void}
    \mid S \rightarrow \text{reflNat } n'
  \}

nl \equiv Z
so
eq n n \equiv \{\text{Void}\}

nl \equiv S
so
eq n n \equiv 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 (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 }}
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 → let p (tag,node) = as in
   case tag of {
      Nil → Void
      Cons → let p (a,as') = node in
         case p a of {
            True → prop A p as
            False → 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
• Implementing general recursion
  Can be difficult to restart evaluation when unfolding a definition.

• We glue together a neutral with its content

\[ x \; t \; \ldots \; [:= v] \]

• 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

\[
eq u v = \{ \{ \text{Void} \} \mid u \equiv v \}
\]
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: \textbf{co-data}

\[
\begin{align*}
\text{stream} & : (A : \text{Type}) \rightarrow \text{Type} \\
\text{stream} & = \lambda A \rightarrow \text{let } \{ \text{Cons} ; A ; \text{case } l \text{ of } \{ \text{Cons } \rightarrow (\text{stream } A) - \} \text{ in } \} \\
\text{zeros} & : \text{stream } \text{Nat} \\
\text{zeros} & = 0, [\text{zeros}] 
\end{align*}
\]
General Boxes

• To compute we need to open a box
  \[ \text{open} \ [ t ] \equiv t \]

• Our boxes are a special case:
  \[ \text{open} \ (\text{case } e \text{ of } \{ \ldots \rightarrow [ t ]\}) \]

• Working with codata

\[
\text{tail} : \text{stream } A \rightarrow \text{stream } A \\
\text{tail} = \lambda xs \rightarrow \text{letp } (tag, \text{node}) = xs \text{ in}
  \text{case } tag \text{ of}
  \{ \text{Cons } \rightarrow \text{letp } (_, \text{tl}) = \text{node } \text{in}
    \text{open } \text{tl} \} \\
\]
More to do

• Integrate meta-variables.
  May have strange interaction with constraints.

• Reflection and generic programming.

• Phase separation and compiler.

• Evidence based optimization.
More to do

• Integrate meta-variables.  
  May have strange interaction with constraints.

• Reflection and generic programming.

• Phase separation and compiler.

• Evidence based optimization.

• Find a good name. Any suggestions?
More to do

• Integrate meta-variables. May have strange interaction with constraints.

• Reflection and generic programming.

• Phase separation and compiler.

• Evidence based optimization.

• Find a good name. Any suggestions?

• Maybe Epigram--