# Refinement Type Inference with Sequential Algorithms

Denis Dancanet

http://www.cs.cmu.edu/~ddr

# What are refinement types?

- ```
  val not = fn x => if x then false else true;

  val map = letrec mapf =
    fn f => fn l => if null l then []
          else (f (hd l)) :: ((mapf f) (tl l))
    in mapf
    end;
  ```

- $true$, $false$ refine $bool$

- $empty$, $one$, $many$ refine $list$

- $not : true \rightarrow false \wedge false \rightarrow true$

  $$\begin{aligned} map : & (\alpha \rightarrow \beta) \rightarrow \ \alpha\ empty \ \rightarrow \ \beta\ empty \ \wedge \\ & (\alpha \rightarrow \beta) \rightarrow \ \alpha\ one \ \rightarrow \ \beta\ one \ \wedge \\ & (\alpha \rightarrow \beta) \rightarrow \ \alpha\ many \ \rightarrow \ \beta\ many. \end{aligned}$$
  ```

# Freeman and Pfenning approach

- Refine datatypes:

$$
\begin{array}{l}
\texttt{datatype}\ \alpha\ \mathit{list}\ \texttt{=}\ \texttt{nil}\ \texttt{|}\ \texttt{cons of}\ \alpha * \alpha\ \mathit{list} \\
\texttt{rectype}\ \alpha\ \mathit{empty}\ \texttt{=}\ \texttt{nil} \\
\quad \texttt{and}\ \alpha\ \mathit{singleton}\ \texttt{=}\ \texttt{cons}\ (\alpha, \texttt{nil}) \\
\quad \texttt{and}\ \alpha\ \mathit{long}\ \texttt{=}\ \texttt{cons}\ (\alpha, \texttt{cons}(\alpha, \alpha\ \top_{list})) \\
\quad \texttt{and}\ \alpha\ \bot_{list}\ \texttt{=}\ \texttt{bottom}\ (\mathit{list})
\end{array}
$$

- Refinement type lattice:

$$
\begin{tikzcd}[arrows=dash]
 & \alpha\ \top_{list} \arrow[dl] \arrow[d] \arrow[dr] & \\
\alpha\ \mathit{empty} \arrow[dr] & \alpha\ \mathit{singleton} \arrow[d] & \alpha\ \mathit{long} \arrow[dl] \\
 & \alpha\ \bot_{list} &
\end{tikzcd}
$$

- Use refinement type inference rules

- Use pending analysis for fixpoints

# Freeman and Pfenning strengths

- Rich language for defining refinement types:

  `datatype` $blist$ `= nil | cons of` $bool\ *\ list$
  `rectype` $bev$ `= cons` $(\top_{bool} *\ bod)$ `| nil` $(runit)$
  `and` $bod$ `= cons` $(\top_{bool} *\ bev)$

- Ability to obtain many interesting refinement types:

$$\begin{aligned} cons : \ & (\alpha * \alpha\ empty) \to \alpha\ singleton\ \wedge \\ & (\alpha * \alpha\ singleton) \to \alpha\ long\ \wedge \\ & (\alpha * \alpha\ long) \to \alpha\ long. \end{aligned}$$

$$\begin{aligned} map : \forall \alpha\beta.\ (\alpha \to \beta) \to\ & \alpha\ empty\ \to \beta\ empty\ \wedge \\ & \alpha\ singleton\ \to \beta\ singleton\ \wedge \\ & \alpha\ long\ \to \beta\ long. \end{aligned}$$

- Higher-order functions pose no special problems

# Freeman and Pfenning weaknesses

- Instantiation of refinement type variables:
  1. cannot use polymorphism and get good refinement types
  2. inefficiency

- ```
  let val not = fn x =>
                    if x then false else true
      val double = fn f => fn x => f (f x)
  in double not
  end
  ```

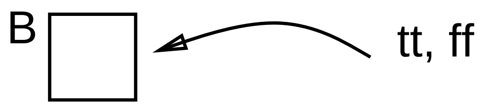
- $double : (\alpha \to \alpha) \to \alpha \to \alpha$

$$
\begin{aligned}
&(true \to true) \to true \to true \ \wedge \\
&(false \to false) \to false \to false \ \wedge \\
&(\top_{bool} \to \top_{bool}) \to \top_{bool} \to \top_{bool} \ \wedge \\
&(\bot_{bool} \to \bot_{bool}) \to \bot_{bool} \to \bot_{bool}
\end{aligned}
$$

- $double\ not : \top_{bool} \to \top_{bool}$
  ```

# Concrete data structures

- A cds is like a record:
  - cells
  - values
  - accessibility (enabling) relation
- Example: BOOL




- Event: cell = value
- State: $\{c_1 = v_1, c_2 = v_2, \ldots\}$
- Example:
  States of BOOL $= \{\{\}, \{B = tt\}, \{B = f\!f\}\}$

# Sequential algorithms

- Sequential algorithm:

  1. Abstract view: sequential function $+$ (sequential) computation strategy
  2. Concrete view: a state of a cds of arrow type (*exponentiation* cds)

- Given two cds's, $M$ and $M'$, the exponentiation cds $M \Rightarrow M'$ is:

  1. the cells are of the form $xc'$, where $x$ is a state of $M$ and $c'$ is a cell of $M'$
  2. the values are of the form *valof* $c$ where $c$ is a cell of $M$, or *output* $v'$ if $v'$ is a value of $M'$

- A state of $M \Rightarrow M'$ is a sequential algorithm

- NOT $\in BOOL \Rightarrow BOOL$

- NOT $= \{\{\}B = valof\ B,$
  $\{B = tt\}B = output\ f\!f,$
  $\{B = f\!f\}B = output\ tt\}$

# CDS0

- Direct implementation of an intensional semantics (sequential algorithms on cds)
- CDS0 was meant to be “assembly” for a higher-level language called CDS
- *Categorical* combinators: application, composition, curry, uncurry, fix, pair, and product
- CDS0 is: lazy, polymorphic, higher-order, functional
- Evaluation model is demand-driven
- Lazy evaluation model allows computing with infinite structures
- Originalities:
  - uniformity of types
  - full abstraction
  - semantics-manipulating programs

# CDS0 Examples

```
let bool =
  dcds
    cell B values tt,ff
  end;

 let not = algo
     request B do
       valof B is
         tt : output ff
         ff : output tt
       end
     end
   end;

 let not_state =
   {{}B     = valof B,
    {B=tt}B = output ff,
    {B=ff}B = output tt};
```

# Types in CDS0

- Original meaning: $x : \tau$ iff $x \in D(\tau)$.

- ```
  let point = dcds
    cell X values [..]
    cell Y values [..]
  end;

  let cPoint = dcds
    cell X values [..]
    cell Y values [..]
    cell C values red, green, blue
  end;
  ```

- Make subtyping in CDS0 like record subtyping

- Computational model in CDS0 is that of incrementally growing a state, which implies:
  $\{$`C = red`$\}$ `:  cPoint`

- Need intersection types, polymorphism, overloading
  ```

# Subtyping

- Two kinds: by extension ($<_e$), by partition ($\leq_p$)

- $cPoint <_e point$

- `let true = dcds cell B values tt end;`
  `let false = dcds cell B values ff end;`

- $true \leq_p bool, \; false \leq_p bool$

- Typing judgments: $x : \tau$ iff $x \in D(\sigma)$ and $\sigma \leq \tau$

- **Theorem:** Incremental computation does not increase the minimal type of a ground state.

- Subtyping is decidable (cds are equivalent to infinite regular trees)

# Type inference

- Collect global input/output information:

| *not* | |
|---|---|
| input | output |
| `B = tt, ff` | `B = tt, ff` |

$not : bool \rightarrow bool$

- Apply inference rules for combinator expressions:

$$(\textsc{App}) \qquad \frac{a : \sigma \rightarrow \tau \quad b : \delta}{a.b : \tau} \quad \text{if } \delta \leq \sigma$$

- Example:

```
not . {B=tt} : bool
```

- **Theorem:** Type inference is sound.

# Refinement types

- Are the subtypes by partition ($\leq_p$)

- Example: integer lists

```
letrec intlist = dcds
  cell EMPTY values true, false
  graft (int.l) access EMPTY = false
  graft (intlist.l) access EMPTY=false
end;

# show more 3 intlist;
{
EMPTY values true, false,
(EMPTY.l) values true, false
                       access EMPTY=false,
((EMPTY.l).l) values true, false
                   access (EMPTY.l)=false,
(N.l) values [..] access EMPTY=false,
((N.l).l) values [..] access (EMPTY.l)=false,
(((N.l).l).l) values [..]
                access ((EMPTY.l).l)=false}
```

# Refinement types (cont.)

```
let empty_intlist = dcds
  cell EMPTY values true
end;

let one_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
  cell (EMPTY.l) values true
                         access EMPTY = false
end;

local letrec partial_intlist = ...
in let many_intlist = dcds
  cell EMPTY values false
  cell (N.l) values [..] access EMPTY = false
  cell (EMPTY.l) values false
                         access EMPTY = false
  cell ((N.l).l) values [..]
                     access (EMPTY.l) = false
  graft (partial_intlist.l)
  end
end;
```

# Refinement type inference

- Collect input/output information for each path through an algorithm:

| $not$ | |
|---|---|
| input | output |
| `B = tt` | `B = ff` |
| `B = ff` | `B = tt` |

$not :_r true \rightarrow false \wedge false \rightarrow true$

- Type-directed AI of combinator expressions:

```
# not | not;
r: /\[false -> false, true -> true]
 : bool -> bool
request? {}B;
--> valof B
request? {B=tt}B;
--> output tt
...
```

- **Theorem:** Refinement type inference is sound.

# A higher-level language

- Lazy, higher-order, polymorphic language:

  ```
  val not = fn x => if x then false else true;

  val map = letrec mapf =
    fn f => fn l => if null l then []
          else (f (hd l)) :: ((mapf f) (tl l))
    in mapf
    end;
  ```

- Compiled to categorical combinators represented by sequential algorithms:

  ```
  $ print not;
  not = curry (cond |
        < < snd, curry (fst) . {B=ff} >,
            curry (fst) . {B=tt} >) . emptyenv
  ```

# Examples

```
# cons;
r: /\[(int * one) -> many,
      (int * many) -> many,
      (int * empty) -> one]
 : (int * intlist) -> intlist

$ val not = fn x => if x then false else true;
r: /\[false -> true, true -> false]
 : bool -> bool

$ val exclmid = fn x => x or (not x);
r: /\[false -> true, true -> true]
 : bool -> bool

$ val map = ...
r: /\[(int -> int) -> many -> many,
      (int -> int) -> one -> one,
      (int -> int) -> empty -> empty]
 : (int -> int) -> intlist -> intlist
```

# More examples

```
$ val l3 = 1 :: (2 :: (3 :: []));
r: many
 : intlist

$ tl l3;
r: many
 : intlist

$ tl (tl l3);
r: one
 : intlist

$ val double = fn f => fn x => f (f x);
r: ('a -> 'a) -> 'a -> 'a
 : ('a -> 'a) -> 'a -> 'a

$ double not;
r: /\[false -> false, true -> true]
 : bool -> bool
```

# And more examples

```
$ val f = fn x => fn y => fn l => (x+y)::l;
r: /\[int -> int -> many -> many,
      int -> int -> one -> many,
      int -> int -> empty -> one]
 : int -> int -> intlist -> intlist
```

```
$ val h = fn x => fn l =>
             if x=3 then l else x::l;
r: int -> intlist -> intlist
 : int -> intlist -> intlist
```

```
$ val f11 = letrec f = fn b => fn g =>
             if g b then true else (f false) g
            in f end;
r: bool -> (bool -> bool) -> bool
 : bool -> (bool -> bool) -> bool
```

# Conclusions

- Strengths:
  - polymorphism poses no problems
  - efficiency (no need to consider all refinements of a type)
  - ability to obtain more precise refinement types in some cases

- Weaknesses:
  - CDS02 operational semantics slow for certain fixpoint computations
  - inability to obtain precise types in some cases (esp. when higher-order inputs are involved)
  - more restrictive language for defining refinement types

# Future work: combined approach?

- Divide program into "chunks"
- Use type-directed AI on the chunks
- Use refinement type inference rules to combine types
- Example: `rev (tl ones)`

```
val ones = Y (letrec f = fn l => 1::(f l)
                  in f end);

$ tl ones;
r: many
 : intlist
```

# Questions and problems

- Refinement type inference for a message passing parallel language
- Use CDS0 to analyze complexity of lazy computation
- Is intensionality inherently expensive to use?
- What is intensional programming good for?

# Cons

```
let cons = algo
  request EMPTY do
    output false
  end
  request (N.l) do
    valof (N.1) is
      $V : output $V
    end
  end
  request (EMPTY.l) do
    valof (EMPTY.2) is
      $B : output $B
    end
  end
  request (((EMPTY.$T).l).l) do
    from {((EMPTY.$T).2)=false} do
      valof (((EMPTY.$T).l).2) is
        $B : output $B
      end
    end
  end
  request (((N.$T).l).l) do
    from {((EMPTY.$T).2)=false} do
      valof (((N.$T).l).2) is
        $V : output $V
      end
    end
  end
end;
```

# Hd and Tl

```
let hd = algo
  request N do
    valof EMPTY is
      false : valof (N.l) is
                $V : output $V
              end
    end
  end
end;

let tl = algo
  request (EMPTY.$T) do
    from {(EMPTY.$T)=false} do
      valof ((EMPTY.$T).l) is
        $B : output $B
      end
    end
  end
  request ((N.$T).l) do
    from {(EMPTY.$T)=false,
          ((EMPTY.$T).l)=false} do
      valof (((N.$T).l).l) is
        $V : output $V
      end
    end
  end
end;
```

# And taster

```
let and_taster =
  algo
    request WHICH_AND do
      valof {}B is
        output tt: output IS_NOT_AN_AND
        output ff: output IS_NOT_AN_AND
        valof (B.1):
          valof {(B.1)=tt}B is
            output tt: output IS_NOT_AN_AND
            output ff: output IS_NOT_AN_AND
            valof (B.2):
              valof {(B.1)=tt,(B.2)=tt}B is
                output ff: output IS_NOT_AN_AND
                output tt:
                  valof {(B.1)=tt,(B.2)=ff}B is
                    output tt: output IS_NOT_AN_AND
                    output ff:
                      valof {(B.1)=ff}B is
                        output tt: output IS_NOT_AN_AND
                        output ff: output IS_LEFT_AND
                        valof (B.2):
                          valof {(B.1)=ff,(B.2)=tt}B is
                            output tt: output IS_NOT_AN_AND
                            output ff:
                              valof {(B.1)=ff,(B.2)=ff}B is
                                output tt: output IS_NOT_AN_AND
                                output ff: output IS_LEFT_STRICT_AND
                              end
                          end
                      end
                  end
              end
          end
        valof (B.2): ...
```

# And taster's refinement type

```
r: /\[/\[(false * false) -> false, (true * true) -> true,
         (true * false) -> false, (false * true) -> false] -> is_and_type,
      ((bool * bool) -> true) -> is_not_and_type,
      /\[(false * false) -> true, (true * true) -> true,
         (true * false) -> false, (false * true) -> false] -> is_not_and_type,
     ((bool * bool) -> false) -> is_not_and_type,
     /\[(true * false) -> true, (true * true) -> true,
        (false * true) -> false] -> is_not_and_type,
     ((true * bool) -> true) -> is_not_and_type,
     /\[(bool * false) -> false, (true * true) -> true,
        (false * true) -> false] -> is_and_type,
     ((true * bool) -> false) -> is_not_and_type,
     /\[(bool * false) -> true, (true * true) -> true,
        (false * true) -> false] -> is_not_and_type,
     ((true * true) -> false) -> is_not_and_type,
     /\[(false * true) -> true, (true * true) -> true] -> is_not_and_type,
     /\[(true * false) -> true, (true * true) -> true] -> is_not_and_type,
     ((bool * true) -> false) -> is_not_and_type,
     /\[(false * bool) -> true, (true * true) -> true,
        (true * false) -> false] -> is_not_and_type,
     ((bool * true) -> true) -> is_not_and_type,
     /\[(false * bool) -> false, (true * true) -> true,
        (true * false) -> false] -> is_and_type,
     /\[(false * false) -> false, (true * true) -> true,
        (false * true) -> false, (true * false) -> false] -> is_and_type,
     /\[(false * true) -> true, (true * true) -> true,
     (true * false) -> false] -> is_not_and_type,
     /\[(false * false) -> true, (true * true) -> true,
        (false * true) -> false, (true * false) -> false] -> is_not_and_type]
 : ((bool * bool) -> bool) -> and_type
```