

A Dependently Typed Core Language

Robert Reitmeier

`rxr@cs.nott.ac.uk`

University of Nottingham

Project "Dependent Types for Haskell"

- Joint work with Simon Peyton Jones, much inspired by Epigram
- Haskell is evolving: Proof of concept for dependent types
- Compiler for Epigram?!

Overview

1. Basic Design Issues
2. Example Code
3. Screenshots
4. Future Work

Basic Design Issues

- Haskellish syntax
- Old school, non-interactive programming
- General recursion \implies functions possibly non-total (Problems?!)
- Non-termination \leftrightarrow proof erasure tradeoff

Primitive Types

- Finite types (nullary constructors), e.g. $\{\text{Zero}, \text{Succ}\}$
- Dependent pairs, e.g.

$$\text{nat} := \left(l : \{\text{Zero}, \text{Succ}\} ; \begin{cases} \mathbb{1} & \text{if } l = \text{Zero} \\ \text{nat} & \text{if } l = \text{Succ} \end{cases} \right)$$

- Dependent functions, e.g. $\text{append} :$
 $n_1 : \text{nat} \rightarrow n_2 : \text{nat} \rightarrow \text{vec } n_1 \rightarrow \text{vec } n_2 \rightarrow \text{vec } (n_1 + n_2)$
- Type of types \star
- Equality type $a = b$

Heterogenous Equality

$$\frac{s_0 : S_0 \quad s_1 : S_1}{s_0 = s_1 : \star} \text{EQFORM}$$

$$\frac{s : S}{\text{Refl}_s : s = s} \text{EQINTRO}$$

Coercions

to state equality across propositionally equal type borders:

$$\frac{s : S_0 \quad p : S_0 = S_1}{s |p\rangle : S_1} \text{ COERCION[EQ]}$$
$$s \llbracket p \rrbracket : s = s |p\rangle$$

Example:

$$\text{append } nil \ v_2 \ := \ v_2$$

but $v_2 : \text{vec } l_2$ instead of $\text{vec } (0 + l_2)$.

Given $p : l_2 = 0 + l_2$, we can coerce v_2 to

$$v_2 |p\rangle : \text{vec } (0 + l_2)$$

Example Code: leqNat

$$\text{leqNat } x \ y \ := \begin{cases} 1 & \text{if } x \leq y, \\ \emptyset & \text{otherwise.} \end{cases}$$

```
1  leqNat : x:nat -> y:nat -> * ;
2  leqNat x y = let (lx,x') = x in case lx of
3      Zero -> { () } ;
4      Succ -> let (ly,y') = y in case ly of
5          Zero -> { } ;
6          Succ -> leqNat x' y' ; ;
7  .
```


Example Code: sub

Subtracts two natural numbers, given a proof that $y \leq x$:

```
1  sub : x:nat -> y:nat -> p:leqNat y x -> nat
2  sub x y p = let (ly,y') = y in case ly of
3      Zero -> x ;
4      Succ -> let (lx,x') = x in case lx of
5          Zero -> case p of ;
6          Succ -> sub x' y' p ; ;
7  .
```

(BTW: Proof argument p can be *erased* at runtime)

Type Checker GUI

The screenshot shows a window titled "tc" containing a tree view of a type checking expression. The tree structure is as follows:

- sub
 - tcExprX let (ly,y') = y in case ly of Zero -> x ; Succ -> let (lx,x') = x in case lx of Zero -> case p of ; Succ -> sub x' y' p ; nat
 - matchX (ly,y') y
 - > [y=(ly,y')]
 - infTypeX y
 - > nat
 - matchTyX (ly,y') nat
 - > [ly:{Zero,Succ}, y':case l of Zero -> unit ; Succ -> nat ;]
 - tcExprX case ly of Zero -> x ; Succ -> let (lx,x') = x in case lx of Zero -> case p of ; Succ -> sub x' y' p ; nat
 - evalX {Zero,Succ}
 - > {Zero,Succ}
 - tcExprX Zero {Zero,Succ}
 - tcExprX x nat
 - tcExprX Succ {Zero,Succ}
 - tcExprX let (lx,x') = x in case lx of Zero -> case p of ; Succ -> sub x' y' p ; nat (selected)
 - matchX (lx,x') x

Below the tree view are two tables:

Var	Value
ly	Succ
l	ly
y	(ly,y')
empty	{}
unit	{() }
nat	(l:{Zero,Succ} ; case l of Zero -> unit ; Succ -> nat ;)
add	{ x -> \ y -> let (l,x') = x in case l of Zero -> y ; Succ -> (Succ,add
vec	{ n -> \ a -> (l:{Nil,Cons} ; case l of Nil -> n = (Zero,()) : nat ; Con
append	{ n1 -> \ n2 -> \ a -> \ v1 -> \ v2 -> let (c,r) = v1 in case c of Nil ->
leqNat	{ x -> \ y -> let (lx,x') = x in case lx of Zero -> unit ; Succ -> let (ly
sub	{ x -> \ y -> \ p -> let (ly,y') = y in case ly of Zero -> x ; Succ -> le

Var	Type
y'	case l of Zero -> unit ; Succ -> nat ;
l	{Zero,Succ}
ly	{Zero,Succ}
p	leqNat y x
y	nat
x	nat
empty	*
unit	*
nat	*
add	x:nat -> y:nat -> nat
vec	n:nat -> a:* -> *

Evaluation GUI

The screenshot shows a window titled 'tc' containing a tree view of a lambda calculus expression. The expression is:

```

evalK sub x y ()
├── evalK sub x y
│   ├── \ p.2 -> let (ly.3,y'.6) = (Succ,(Succ,[Zero,()])) in case ly.3 of Zero -> (Succ,(Succ,(Succ,[Zero,()]))) ; Succ -> let (lx.4,x'.5) = (Succ,(Succ,(Succ,[Zero,()]))) in case lx.4 of Zero -> case () of ; ;
│   └── evalK ()
│       └── -> ()
├── evalK let (ly.3,y'.6) = (Succ,(Succ,[Zero,()]))) in case ly.3 of Zero -> (Succ,(Succ,(Succ,[Zero,()]))) ; Succ -> let (lx.4,x'.5) = (Succ,(Succ,(Succ,[Zero,()]))) in case lx.4 of Zero -> case () of ; ;
│   ├── evalK (Succ,(Succ,[Zero,()])))
│   │   └── -> (Succ,(Succ,[Zero,()])))
│   ├── matchK (ly.3,y'.6) (Succ,(Succ,[Zero,()])))
│   │   └── -> [ ly.3=Succ, y'.6=(Succ,[Zero,()]) ]
│   └── evalK case ly.3 of Zero -> (Succ,(Succ,(Succ,[Zero,()]))) ; Succ -> let (lx.4,x'.5) = (Succ,(Succ,(Succ,[Zero,()]))) in case lx.4 of Zero -> case () of ; ;
│       └── -> (Succ,[Zero,()])
└── -> (Succ,[Zero,()])
    └── -> (Succ,[Zero,()])
        └── -> (Succ,[Zero,()])
    
```

Below the tree view are two tables:

Var	Value	Var	Type
leqNat	{ x -> { y -> let (lx,x') = x in case lx of Zero -> unit ; Succ -> let (ly	empty	*
sub	{ x -> { y -> { p -> let (ly,y') = y in case ly of Zero -> x ; Succ -> le	unit	*
eqSucc	{ n -> { m -> { p -> case p of Refl -> Refl ;	nat	*
addxZ...	{ x -> let (lx,x') = x in case lx of Zero -> case x' of () -> Refl ; ; Succ	add	x:nat -> y:nat -> nat
zero	(Zero,())	vec	n:nat -> a:* -> *
one	(Succ,zero)	append	n1:nat -> n2:nat -> a:* -> v1:vec n1 a -> v2:vec n2 a -> vec (adc
two	(Succ,one)	leqNat	x:nat -> y:nat -> *
three	(Succ,two)	sub	x:nat -> y:nat -> p:leqNat y x -> nat
main	let x = three in let y = two in sub x y ()	eqSucc	n:nat -> m:nat -> p:n = m -> (Succ,n) = (Succ,m)

Future Work

- Complete the implementation, give more examples
- Prove soundness of the typing algorithm
- High level compiler
- Proof erasure
- Integration in Haskell

End