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 $\Longrightarrow$ functions possibly non-total (Problems?!)
- Non-termination $\leftrightarrow$ proof erasure tradeoff

Primitive Types

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

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

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

Heterogenous Equality

$$\frac{s_0 : S_0 \qquad s_1 : S_1}{s_0 = s_1 : \star}\ \textsc{EqForm}$$

$$\frac{s : S}{\mathrm{Refl}_s : s = s}\ \textsc{EqIntro}$$

Coercions

to state equality across propositionally equal type borders:

$$\frac{s : S_0 \qquad p : S_0 = S_1}{s \,|p\rangle : S_1} \textsc{Coercion[Eq]}$$

$$s \,[\![p\rangle\!\rangle : s = s\,|p\rangle$$

Example:

$$\mathrm{append}\ \mathit{nil}\ v_2 \quad := \quad v_2$$

but $v_2 : \mathrm{vec}\ l_2$ instead of $\mathrm{vec}\ (0 + l_2)$.
Given $p \;:\; l_2 = 0 + l_2$, we can coerce v_2 to

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

Example Code: `leqNat`

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

```
leqNat : x:nat -> y:nat -> * ;
leqNat x y = let (lx,x') = x in case lx of
    Zero -> { () } ;
    Succ -> let (ly,y') = y in case ly of
        Zero -> { } ;
        Succ -> leqNat x' y' ; ;
.
```

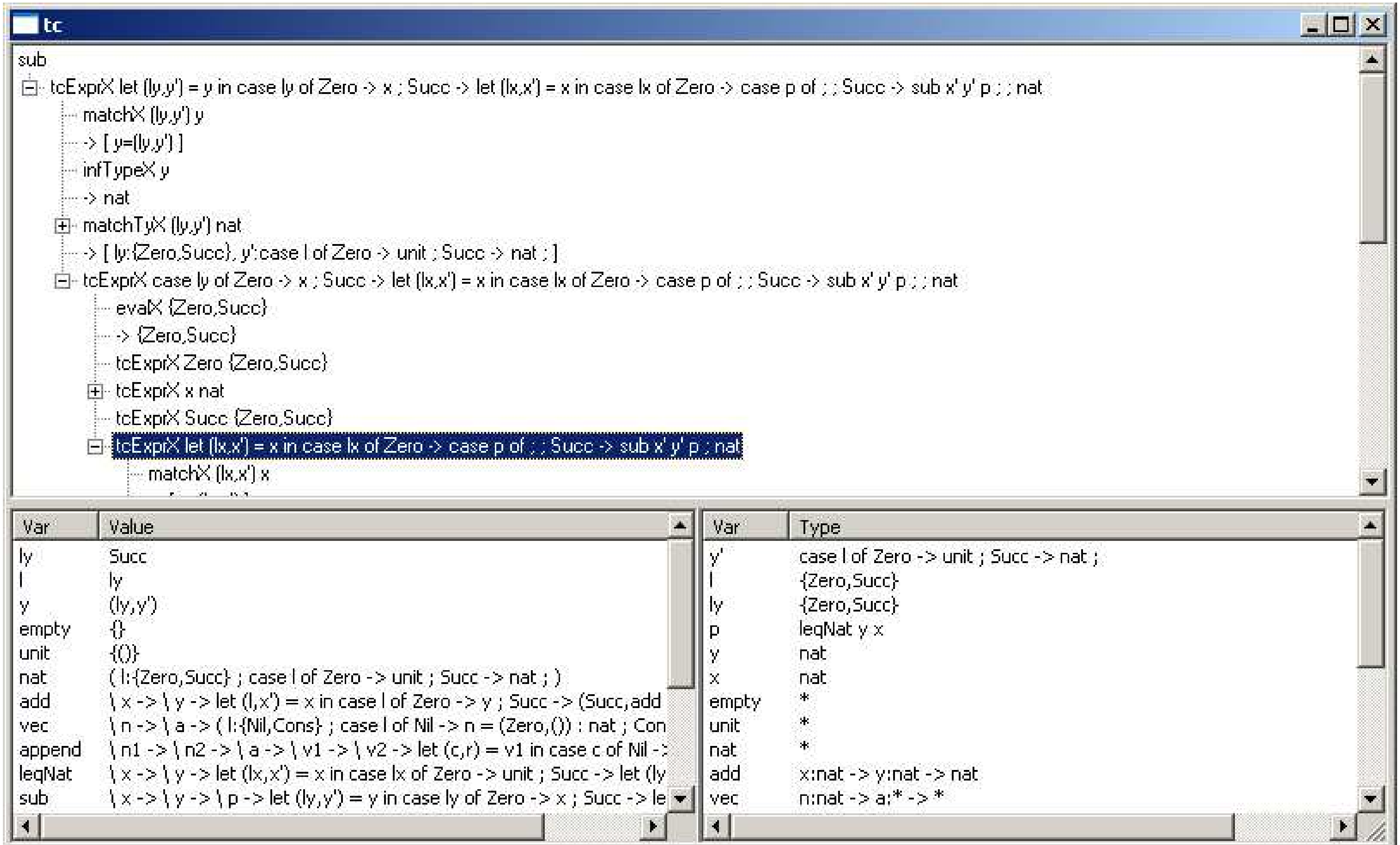
Example Code: sub

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

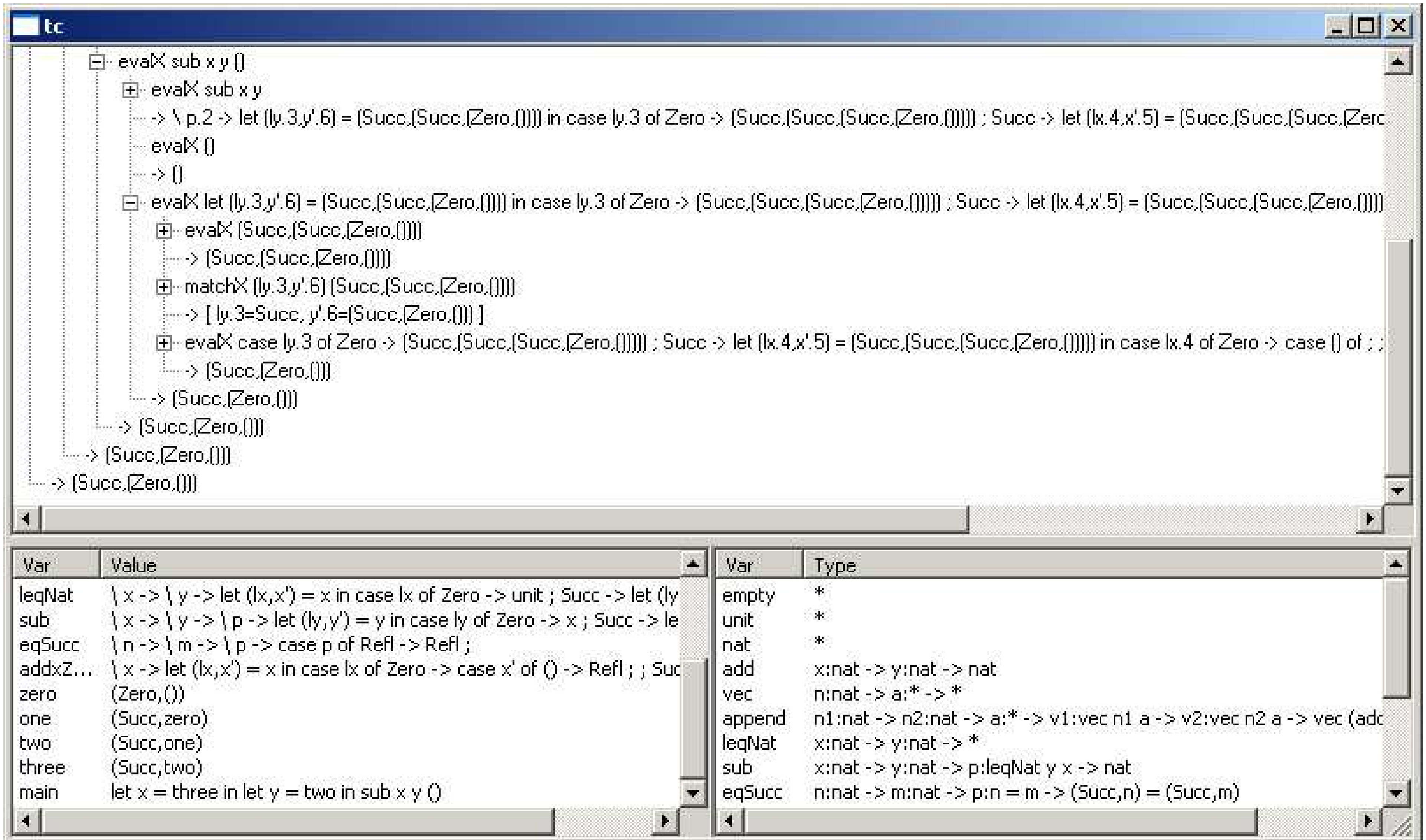
```
sub : x:nat -> y:nat -> p:leqNat y x -> nat
sub x y p = let (ly,y') = y in case ly of
    Zero -> x ;
    Succ -> let (lx,x') = x in case lx of
        Zero -> case p of ;
        Succ -> sub x' y' p ; ;
.
```

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

Type Checker GUI



Evaluation GUI



Future Work

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

End