An Introduction to Type Theory

Part 3

Tallinn, September 2003
with cartoons by Conor McBride

http://www.cs.nott.ac.uk/~txa/tallinn/

Thorsten Altenkirch
University of Nottingham

The established social order

terms	types
do all the work	are never around when there is work to be done
engage in criminal activity	commit no crime
can be stopped and searched	cannot be investigated.
belong to and are hold in check by types	

Ersatz dependent types

- In modern type systems types have to do some work
- Polymorphic types can be use to represent square matrices

$$
\begin{aligned}
\mathrm{Matrix}\ a &= \mathrm{Matrix}'\ \mathrm{Nil}\ a \\
\mathrm{Matrix}'\ t\ a &= \mathrm{Zero}\ (t\ (t\ a)) \mid \mathrm{Succ}\ (\mathrm{Cons}\ t)\ a \\
\mathrm{Nil}\ a &= \mathrm{Nil} \\
\mathrm{Cons}\ t\ a &= \mathrm{Cons}\ a\ (t\ a)
\end{aligned}
$$

- Conor showed in *Faking it* how to use the logic programming of Haskell's class system to simulate some usages of dependent types.

Breaking the old social order

- Data is validated wrt other data
- If types are to capture the validity of data, we must let them depend on terms.

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq
- Emphasis on safe and efficient execution
- Using **epigram** currently developed by Conor, using ideas from
  - LEGO / OLEG
  - ALF

nth — no dependent types

$$\text{let}\ \frac{\mathrm{A} \in * \quad l \in \mathrm{List}\ \mathrm{A} \quad n \in \mathrm{Nat}}{\mathrm{nth}\ \mathrm{A}\ l\ n \in \mathrm{A}}$$

$$\mathrm{nth}\ \mathrm{A}\ l\ n \mapsto ? \qquad \begin{array}{ll} \mathrm{nth}\ \mathrm{nil}\ n & \mapsto ? \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ n & \mapsto ? \end{array} \qquad \begin{array}{ll} \mathrm{nth}\ \mathrm{nil}\ n & \mapsto ? \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ 0 & \mapsto ?a \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ (\mathrm{s}\ n) & \mapsto ?\mathrm{nth}\ as \end{array}$$

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information

nth is not good

$$\text{let}\ \frac{l \in \mathrm{List\ A} \qquad n \in \mathrm{Nat}}{\mathrm{nth}\ l\ n \in \mathrm{A}} \qquad \begin{array}{ll} \mathrm{nth\ nil}\ n & \mapsto ? \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ 0 & \mapsto a \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ (\mathrm{s}\ n) & \mapsto \mathrm{nth}\ as\ n \end{array}$$

- The function nth is partial

$$\mathrm{nth}\ 3\ [1, 2]$$

  leads to a runtime error.
- Reason: The type of nth is not informative enough.

data types

$$\text{data}\ \frac{}{\mathrm{Nat} \in *}\ \text{where} \qquad \frac{}{0 \in \mathrm{Nat}} \qquad \frac{n \in \mathrm{Nat}}{\mathrm{s}\ n \in \mathrm{Nat}}$$

$$\text{data}\ \frac{\mathrm{A} \in *}{\mathrm{List\ A} \in *}\ \text{where}$$

$$\frac{\mathrm{A} \in *}{\mathrm{nil}_{\mathrm{A}} \in \mathrm{List\ A}} \qquad \frac{\mathrm{A} \in * \qquad a \in \mathrm{A} \qquad as \in \mathrm{List\ A}}{\mathrm{cons}_{\mathrm{A}}\ a\ as \in \mathrm{List\ A}}$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0'_n \in \mathrm{Fin}\ \ (\mathrm{s}\ n)}\quad \frac{n \in \mathrm{Nat} \qquad i \in \mathrm{Fin}\ \ n}{\mathrm{s}'_n\ i \in \mathrm{Fin}\ \ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \qquad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{n \in \mathrm{Nat} \quad a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}_n\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{n \in \mathrm{Nat} \quad l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}_n\ l\ i \in \mathrm{A}}$$

$$\mathrm{nth}\ l\ n \mapsto ? \quad \begin{array}{ll} \mathrm{nth}\ l\ 0' \mapsto ? & \mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' \mapsto ? \\ \mathrm{nth}\ l\ (\mathrm{s}'\ i) \mapsto ? & \mathrm{nth}\ l\ (\mathrm{s}'\ i) \mapsto ? \end{array} \quad \begin{array}{ll} \mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' \mapsto ? & \mathrm{nth}\ (\mathrm{vcons} \\ \mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ (\mathrm{s}'\ i) \mapsto ? & \mathrm{nth}\ (\mathrm{vcons} \end{array}$$

- nth is a total function.
- $\mathrm{nth}\ 3\ [1, 2]$ is not well-typed.

Verify

- How can we use nth on lists of unknown length user input,. . . ?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{llll} \mathrm{verify}\ n\ i \mathrm{verify}\ 0\ i & \mapsto & ?\mathrm{nothing} & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ i \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & ?\mathrm{just}\ 0' & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \mapsto \| & ?\mathrm{verify}\ n\ i & \mapsto\ ? \\ & | & \mathrm{nothing} & \mapsto\ ?\mathrm{nothing} \\ & | & \mathrm{just}\ i & \mapsto\ ?\mathrm{just}\ (\mathrm{s}'\ i) \end{array}$$

Going further

- The type of verify is not informative enough for some of its potential applications.
- How is $\mathrm{just}\ j \equiv \mathrm{verify}\ n\ i$ related to j?
- When does verify return $\mathrm{nothing}$?

verify **improved.**

$$\text{let}\ \frac{i \in \mathrm{Fin}\ \ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{llllll}
\mathrm{verify}\ n\ i \mathrm{verify}\ 0\ i & \mapsto & ?\mathrm{tooBig}\ 0\ i & & \\
\mathrm{verify}\ (\mathrm{s}\ n)\ i \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & ?\mathrm{bound}\ n\ 0' & & \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \mapsto \| & ?\mathrm{verify}\ n\ i & \mapsto & ? \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ (n{+}i)) & | & \mathrm{tooBig}\ n\ i & \mapsto & ?\mathrm{tooBig}\ (\mathrm{s}\ n)\ i \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{val}\ i) & | & \mathrm{bound}\ n\ i & \mapsto & ?\mathrm{bound}\ (\mathrm{s}\ n)\ (\mathrm{s}'\ i)
\end{array}$$

Definitional equality

- The typing of verify depends on the equations:

$$\begin{array}{rcl} 0+n & \equiv & n \\ (\mathrm{s}\ m)+n & \equiv & \mathrm{s}\ (m+n) \end{array}$$

- This equations need to be true *definitionally*.
- If we need $n+0 = n$ we have to use propositional equality.

Propositional equality

$$\text{data}\ \frac{\mathrm{A} \in * \quad a, b \in \mathrm{A}}{a{=}b \in *}\ \text{where}\quad \frac{a \in \mathrm{A}}{\mathrm{refl}\ a \in a{=}a}$$

$$\text{let}\ \frac{n \in \mathrm{Nat}}{\mathrm{add}_0\ n \in n{+}0{=}n}\quad \begin{array}{llllll} \mathrm{add}_0\ 0 & \mapsto & \mathrm{refl}\ 0 & & & \\ \mathrm{add}_0\ (\mathrm{s}\ n) & \| & n+0 & & & \\ & | & m & \| & \mathrm{add}_0\ n & \\ & | & n & | & \mathrm{refl} & \mapsto\ \mathrm{refl} \end{array}$$

$$\text{let}\ \frac{q \in a{=}b \quad \mathrm{P} \in \mathrm{A} \to * \quad x \in \mathrm{P}\ a}{\mathrm{subst}\ q\ \mathrm{P}\ x \in \mathrm{P}\ b}\quad \mathrm{subst}\ (\mathrm{refl}\ a)\ \mathrm{P}\ x \mapsto x$$

Problems with $=$

- Programs cluttered with coercions.
- Programming requires theorem proving.
- Equality on functions is not extensional, i.e.

$$\text{let}\ \frac{f, g \in \mathrm{A} \to \mathrm{B} \quad p \in \forall x \in \mathrm{A}. f\ x = g\ x}{\mathrm{ext}\ p \in f = g}$$

cannot be derived.

Solutions ?

- In many cases the need for propositional equality can be avoided.
- DML shows that many equalities needed in programming can be proven automatically.
  Proposal: Integrate an extensible constraint prover into the elaboration process.
- The problem with extensional equality can be overcome using a different approach to equality.

Eval

- Implement an evaluator for a simply typed object language.
- We use type-checking to avoid run-time errors.
- First we implement a simply typed version.
- Then a dependently typed version, exploiting the *verify pattern*.

The object language

data $\dfrac{}{\mathrm{Val} \in *}$ where $\dfrac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{Val}}$ $\dfrac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{Val}}$

data $\dfrac{}{\mathrm{Tm} \in *}$ where

$$\dfrac{v \in \mathrm{Val}}{\mathrm{tval}\ v \in \mathrm{Tm}} \quad \dfrac{t, u_0, u_1 \in \mathrm{Tm}}{\mathrm{tif}\ t\ u_0\ u_1 \in \mathrm{Tm}} \quad \dfrac{t, u, \in \mathrm{Tm}}{\mathrm{tadd}\ t\ u \in \mathrm{Tm}}$$

Object types

data $\dfrac{}{\mathrm{Ty} \in *}$ where $\dfrac{}{\mathrm{nat} \in \mathrm{Ty}}$ $\quad$ $\dfrac{}{\mathrm{bool} \in \mathrm{Ty}}$

let $\dfrac{t \in \mathrm{Tm}}{\mathrm{verify}\ t \in \mathrm{Maybe}\ \ \mathrm{Ty}}$ $\quad \cdots$

Eval — simply typed

let $\dfrac{t \in \mathrm{Tm}}{\mathrm{eval}\ t \in \mathrm{Val}}$

$$
\begin{array}{llllllll}
\mathrm{eval}\ (\mathrm{tval}\ v) & \mapsto & v \\
\mathrm{eval}\ (\mathrm{tif}\ t\ u_0\ u_1) & \| & \mathrm{eval}\ t \\
 & | & \mathrm{vnat}\ n & \mapsto & ? \\
 & | & \mathrm{vbool\ true} & \mapsto & \mathrm{eval}\ u_0 \\
 & | & \mathrm{vbool\ false} & \mapsto & \mathrm{eval}\ u_1 \\
\mathrm{eval}\ (\mathrm{tadd}\ t\ u) & \| & \mathrm{eval}\ t \\
 & | & \mathrm{vnat}\ m & \| & \mathrm{eval}\ u \\
 & & & | & \mathrm{vnat}\ n & \mapsto & \mathrm{vnat}(m+n) \\
 & & & | & \mathrm{vbool}\ b & \mapsto & ? \\
 & | & \mathrm{vbool}\ b & \mapsto & ?
\end{array}
$$

Safe eval ?

$$\text{let } \frac{t \in \mathrm{Tm}}{\mathrm{seval}\ t \in \mathrm{Maybe}\ \ \mathrm{Val}}$$

$$\begin{array}{llllll} \mathrm{seval}\ t & \| & \mathrm{verify}\ t & & \\ & | & \mathrm{just}\ u & \mapsto & \mathrm{just}\ (\mathrm{eval}\ u) \\ & | & \mathrm{nothing} & \mapsto & \mathrm{nothing} \end{array}$$

Safe eval ?

- We know that seval will never crash . . .
- . . . but the compiler doesn't!
- seval is inefficient:
  - Values carry tags at runtime
  - eval checks the tags

Object language using dependent types

$$\text{data}\ \frac{a \in \mathrm{Ty}}{\mathrm{TVal}\ a \in *}\ \text{where}\quad \frac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{TVal\ nat}}\quad \frac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{TVal\ bool}}$$

$$\text{data}\ \frac{a \in *}{\mathrm{TTm}\ a \in *}\ \text{where}$$

$$\frac{v \in \mathrm{TVal}\ a}{\mathrm{tval}\ v \in \mathrm{TTm}\ a}\quad \frac{t \in \mathrm{TTm\ bool}\quad u_0, u_1 \in \mathrm{TTm}\ a}{\mathrm{tif}\ t\ u_0\ u_1 \in \mathrm{TTm}\ a}\quad \frac{t, u, \in \mathrm{TTm\ nat}}{\mathrm{tadd}\ t\ u \in \mathrm{TTm\ nat}}$$

Eval — dependently

$$\text{let}\ \frac{t \in \mathrm{TTm}\ a}{\mathrm{eval}\ t \in \mathrm{Val}\ a}$$

$$\begin{array}{llllllll}
\mathrm{eval}\ (\mathrm{tval}\ v) & \mapsto & v & & & & & \\
\mathrm{eval}\ (\mathrm{tif}\ t\ u_0\ u_1) & \| & \mathrm{eval}\ t & & & & & \\
 & | & \mathrm{vbool\ true} & \mapsto & \mathrm{eval}\ u_0 & & & \\
 & | & \mathrm{vbool\ false} & \mapsto & \mathrm{eval}\ u_1 & & & \\
\mathrm{eval}\ (\mathrm{tadd}\ t\ u) & \| & \mathrm{eval}\ t & & & & & \\
 & | & \mathrm{vnat}\ m & \| & \mathrm{eval}\ u & & & \\
 & & & | & \mathrm{vnat}\ n & \mapsto & \mathrm{vnat}(m+n) &
\end{array}$$

Safe eval !

$$\text{let}\ \frac{t \in \mathrm{TTm}\ a}{\mathrm{strip}\ t \in \mathrm{Tm}} \quad \cdots$$

$$\text{data}\ \frac{t \in \mathrm{Tm}}{\mathrm{Verify}\ t \in \mathrm{Ty}}\ \text{where}\ \frac{}{\mathrm{error} \in \mathrm{Verify}\ t} \quad \frac{t \in \mathrm{TTm}\ a}{\mathrm{ok}\ t \in \mathrm{Verify}\ (\mathrm{strip}\ t)}$$

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{verify}\ t \in \mathrm{Verify}\ t} \quad \cdots$$

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{seval}\ t \in \mathrm{Maybe}\ (\Sigma_{a \in \mathrm{Ty}} \mathrm{Val}\ a)}$$

$$\begin{array}{lllll} \mathrm{seval}\ t & \| & \mathrm{verify}\ t & & \\ & | & \mathrm{just}\ u & \mapsto & \mathrm{just}\ (\mathrm{eval}\ u) \\ & | & \mathrm{error} & \mapsto & \mathrm{nothing} \end{array}$$

Safe eval !

- The compiler knows that seval will never crash.
- We can generate a certificate (proof carrying code).
- seval is efficient:
  - No tags at runtime.
  - No checking.

Generic equality

- Implement a generic equality function for non nested, concrete data types
- Usually requires a language extension (Generic Haskell)
- Topic developed further in our (Conor and me) paper: *Generic programming within dependently typed programming* Working Conference on Generic Programming 2002

Codes and data

$$\text{data} \; \frac{}{\mathrm{Ty} \in *} \; \text{where}$$

$$\frac{}{\mathrm{unit} \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \quad \frac{}{\mathrm{rec} \in \mathrm{Ty}}$$

$$\text{data} \; \frac{r, a \in \mathrm{Ty}}{\mathrm{Val}\ r\ a \in *} \; \text{where}$$

$$\frac{}{\mathrm{void} \in \mathrm{Val}\ r\ \mathrm{unit}} \quad \frac{x \in \mathrm{Val}\ r\ a \quad y \in \mathrm{Val}\ r\ b}{\mathrm{pair}\ x\ y \in \mathrm{Val}\ r\ (\mathrm{prod}\ a\ b)}$$

$$\frac{x \in \mathrm{Val}\ a}{\mathrm{inl}\ x \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)} \quad \frac{y \in \mathrm{Val}\ b}{\mathrm{inr}\ y \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)}$$

$$\frac{x \in \mathrm{Val}\ r\ r}{\mathrm{in}\ x \in \mathrm{Val}\ r\ r}$$

Example

$$\text{let}\ \frac{a \in \mathrm{Ty}}{\mathrm{Data}\ a \in \mathrm{Ty}} \quad \mathrm{Data}\ a \mapsto \mathrm{Val}\ a\ a$$

$$\text{let}\ \frac{}{\mathrm{nat} \in \mathrm{Ty}} \quad \mathrm{nat} \mapsto \mathrm{sum\ unit\ rec}$$

$$\text{let}\ \frac{}{\mathrm{zero} \in \mathrm{Data\ nat}} \quad \mathrm{zero} \mapsto \mathrm{inl\ void}$$

$$\text{let}\ \frac{n \in \mathrm{Data\ nat}}{\mathrm{succ}\ n \in \mathrm{Data\ nat}} \quad \mathrm{succ}\ n \mapsto \mathrm{inr}\ (\mathrm{in}\ n)$$

Generic equality

$$\text{let}\ \frac{x, y \in \mathrm{Val}\ r\ t}{\mathrm{eq}\ x\ y \in \mathrm{Bool}}$$

$$\begin{array}{lllll}
\mathrm{eq} & \mathrm{void} & \mathrm{void} & \mapsto & \mathrm{true} \\
\mathrm{eq} & (\mathrm{pair}\ x\ y) & (\mathrm{pair}\ x'\ y') & \mapsto & (\mathrm{eq}\ x\ x')\ \&\&\ (\mathrm{eq}\ y\ y') \\
\mathrm{eq} & (\mathrm{inl}\ x) & (\mathrm{inl}\ x') & \mapsto & \mathrm{eq}\ x\ x' \\
\mathrm{eq} & (\mathrm{inl}\ x) & (\mathrm{inr}\ y) & \mapsto & \mathrm{false} \\
\mathrm{eq} & (\mathrm{inr}\ y) & (\mathrm{inl}\ x) & \mapsto & \mathrm{false} \\
\mathrm{eq} & (\mathrm{inr}\ y) & (\mathrm{inr}\ y') & \mapsto & \mathrm{eq}\ y\ y' \\
\mathrm{eq} & (\mathrm{in}\ x) & (\mathrm{in}\ x') & \mapsto & \mathrm{eq}\ x\ x'
\end{array}$$

dvantages of Dependently Typed Programmin

- Avoidance of run-time errors
- More efficient code (elimination of tags)
- Extensions of Type System as library
- Easier to reason about
- Already comes with a verification language (totality checker)
- Safety constraints can be checked statically (safe mobile code)

Important issues

- Definitional equality should be well behaved
- Inductive families have to be supported
- Type inference is generalized by elaboration
  Extensible elaboration ?
- Programs are constructed interactively,
  starting with the type as a partial specification