# Typed Tagless Final Interpreters

Stuart Terrett

May 10, 2020

# Preamble

**module** *TTFI* **where**

# Typed

The object language is *typed*

# Typed

The object language is *typed*

$$\textit{add}\ (\textit{int}\ 2)\ (\textit{bool}\ 3)$$

# Typed

The object language is *typed*

*add* (*int* $2$) (*bool* $3$)

will not compile

# tagless

The object language terms are not wrapped in a sum type with type tags

# tagless

The object language terms are not wrapped in a sum type with type tags

$$
\begin{aligned}
&\textbf{data}\ \mathit{Term} = \\
&\quad \mathit{TInt}\ \mathit{Int} \\
&\quad \mid \mathit{TBool}\ \mathit{Bool} \\
&\quad \mid \mathit{TStr}\ \mathit{String}
\end{aligned}
$$

# final

Terms of the object language are represented as expressions in the metalanguage

# final

Terms of the object language are represented as expressions in the metalanguage

$$
\begin{aligned}
&\mathit{twoPlusTwo} :: (\mathit{AddSym}\ \mathit{repr}) \Rightarrow \mathit{repr}\ \mathit{Int} \\
&\mathit{twoPlusTwo} = \mathit{add}\ (\mathit{int}\ 2)\ (\mathit{int}\ 2)
\end{aligned}
$$

# final

Terms of the object language are represented as expressions in the metalanguage

$$\begin{aligned} &\mathit{twoPlusTwo} :: (\mathit{AddSym}\ \mathit{repr}) \Rightarrow \mathit{repr}\ \mathit{Int} \\ &\mathit{twoPlusTwo} = \mathit{add}\ (\mathit{int}\ 2)\ (\mathit{int}\ 2) \end{aligned}$$

Not abstract syntax

# interpreter

The syntax is a typeclass

$$
\begin{aligned}
&\textbf{class } \mathit{AddSym}\ \mathit{repr}\ \textbf{where}\\
&\quad \mathit{int} :: \mathit{Int} \to \mathit{repr}\ \mathit{Int}\\
&\quad \mathit{add} :: \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int}
\end{aligned}
$$

# interpreter

The semantics are an instance of the typeclass

$$\textbf{newtype}\ R\ a = R\ \{unR :: a\}$$

$$\textbf{instance}\ AddSym\ R\ \textbf{where}$$
$$\quad int = R$$
$$\quad add\ (R\ x)\ (R\ y) = R\ \$\ x + y$$

## extensible

Adding interpretations

```
newtype S a = S {unS :: Int → String}
instance AddSym S where
  int = S ∘ const ∘ show
  add (S x) (S y) = S $ λc →
    "(" <> x c <> " + " <> y c <> ")"
```

# extensible

Adding operations

$$
\begin{array}{l}
\textbf{class } \mathit{MultSym}\ \mathit{repr}\ \textbf{where} \\
\quad \mathit{mul} :: \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int} \\
\textbf{instance } \mathit{MultSym}\ R\ \textbf{where} \\
\quad \mathit{mul}\ (R\ x)\ (R\ y) = R\ \$\ x * y \\
\textbf{instance } \mathit{MultSym}\ S\ \textbf{where} \\
\quad \mathit{mul}\ (S\ x)\ (S\ y) = S\ \$\ \lambda c \to \\
\qquad \texttt{"("} <> x\ c <> \texttt{" * "} <> y\ c <> \texttt{")"}
\end{array}
$$

# adding booleans

**class** *BoolSym repr* **where**
  $bool :: Bool \rightarrow repr\ Bool$
  $lte :: repr\ Int \rightarrow repr\ Int \rightarrow repr\ Bool$
  $when :: repr\ Bool \rightarrow repr\ a \rightarrow repr\ a \rightarrow repr\ a$

# adding booleans

**instance** *BoolSym R* **where**
  $bool = R$
  $lte\ (R\ x)\ (R\ y) = R\ \$\ x \leqslant y$
  $when\ (R\ b)\ (R\ t)\ (R\ f) = R\ \$\ \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ f$

# adding booleans

$$
\begin{array}{l}
\textbf{instance}\ \mathit{BoolSym}\ R\ \textbf{where} \\
\quad \mathit{bool} = R \\
\quad \mathit{lte}\ (R\ x)\ (R\ y) = R \,\$\, x \leqslant y \\
\quad \mathit{when}\ (R\ b)\ (R\ t)\ (R\ f) = R \,\$\, \textbf{if}\ b\ \textbf{then}\ t\ \textbf{else}\ f
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{BoolSym}\ S\ \textbf{where} \\
\quad \mathit{bool} = S \circ \mathit{const} \circ \mathit{show} \\
\quad \mathit{lte}\ (S\ x)\ (S\ y) = S \,\$\, \lambda c \rightarrow \\
\qquad \texttt{"("} <> x\ c <> \texttt{" <= "} <> y\ c <> \texttt{")"} \\
\quad \mathit{when}\ (S\ b)\ (S\ t)\ (S\ f) = S \,\$\, \lambda c \rightarrow \\
\qquad \texttt{"if "} <> b\ c \\
\qquad <> \texttt{"\textbackslash nthen "} <> t\ c \\
\qquad <> \texttt{"\textbackslash n else"} <> f\ c
\end{array}
$$

# adding lambda abstraction

$$
\begin{array}{l}
\textbf{class}\ \mathit{LamSym}\ \mathit{repr}\ \textbf{where} \\
\quad \mathit{lam} :: (\mathit{repr}\ a \to \mathit{repr}\ b) \to \mathit{repr}\ (a \to b) \\
\quad \mathit{app} :: \mathit{repr}\ (a \to b) \to \mathit{repr}\ a \to \mathit{repr}\ b
\end{array}
$$

# adding lambda abstraction

$$
\begin{array}{l}
\textbf{instance}\ \mathit{LamSym}\ R\ \textbf{where} \\
\quad \mathit{lam}\ f = R\ \$\ \mathit{unR} \circ f \circ R \\
\quad \mathit{app}\ (R\ f)\ (R\ a) = R\ \$\ f\ a
\end{array}
$$

# adding lambda abstraction

```
instance LamSym R where
  lam f = R $ unR ∘ f ∘ R
  app (R f) (R a) = R $ f a
```

```
instance LamSym S where
  lam f = S $ λc →
    let
      x = "x" <> show c
    in
      "\\" <> x <> ".\n"
       <> (unS ∘ f ∘ S $ const x) (c + 1)
  app (S f) (S a) = S $ λc →
    "(" <> f c <> " " <> a c <> ")"
```

# adding recursion!!

```
class FixSym repr where
    fix :: (repr a → repr a) → repr a
```

# adding recursion!!

$$
\begin{array}{l}
\textbf{instance}\ \mathit{FixSym}\ R\ \textbf{where} \\
\quad \mathit{fix}\ f = R\ \$\ \mathit{fx}\ (\mathit{unR} \circ f \circ R) \\
\qquad \textbf{where}\ \mathit{fx}\ g = g\ (\mathit{fx}\ g)
\end{array}
$$

# adding recursion!!

**instance** *FixSym R* **where**
  $fix\ f = R\ \$\ fx\ (unR \circ f \circ R)$
    **where** $fx\ g = g\ (fx\ g)$

**instance** *FixSym S* **where**
  $fix\ f = S\ \$\ \lambda c \rightarrow$
    **let**
      $self = \texttt{"self"} <> show\ c$
    **in**
      $\texttt{"(fix "} <> self <> \texttt{".\textbackslash n"}$
      $<> (unS \circ f \circ S\ \$\ const\ self)\ (c + 1) <> \texttt{")"}$

# simply typed lambda calculus with integer and boolean literals

$$
\begin{aligned}
&\textbf{class}\ (\mathit{AddSym\ repr} \\
&\quad ,\mathit{MultSym\ repr} \\
&\quad ,\mathit{BoolSym\ repr} \\
&\quad ,\mathit{LamSym\ repr} \\
&\quad ,\mathit{FixSym\ repr} \\
&\quad ) \Rightarrow \mathit{Symantics\ repr}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Symantics}\ R \\
&\textbf{instance}\ \mathit{Symantics}\ S
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} :: R\ a \to a \\
&\mathit{eval} = \mathit{unR}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{pprint} :: S\ a \to \mathit{String} \\
&\mathit{pprint}\ e = \mathit{unS}\ e\ 0
\end{aligned}
$$

# factorial

$$\begin{aligned}
&\mathit{factorial} :: (\mathit{Symantics}\ \mathit{repr}) \Rightarrow \mathit{repr}\ (\mathit{Int} \to \mathit{Int})\\
&\mathit{factorial} = \mathit{fix}\ (\lambda \mathit{self} \to \mathit{lam}\ (\lambda n \to\\
&\quad \mathit{when}\ (\mathit{lte}\ n\ (\mathit{int}\ 0))\\
&\qquad (\mathit{int}\ 1)\\
&\qquad (\mathit{mul}\ n\ (\mathit{self}\ \text{`}\mathit{app}\text{`}\ (\mathit{add}\ n\ (\mathit{int}\ (-1)))))))
\end{aligned}$$

(hint: now's a good time to `stack ghci src/TTFI.hs`)