Typed Tagless Final Interpreters

Stuart Terrett

May 11, 2020

preamble

```
{-# LANGUAGE GADTs #-}
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*

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

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

$$\begin{array}{l}\textbf{data}\ \mathit{Term} = \\ \quad \mathit{TInt}\ \mathit{Int} \\ \quad \mid \mathit{TBool}\ \mathit{Bool}\end{array}$$

tagless

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Term} = \\
\quad \mathit{TInt}\ \mathit{Int} \\
\quad |\ \mathit{TBool}\ \mathit{Bool}
\end{array}
$$

The object language is "tight": if it compiles, it cannot get stuck

$$
\begin{array}{l}
\mathit{addU} :: \mathit{Term} \to \mathit{Term} \to \mathit{Term} \\
\mathit{addU}\ (\mathit{TInt}\ x)\ (\mathit{TInt}\ y) = \mathit{TInt}\ \$\ x + y \\
\mathit{addU}\ \_\ \_ = \mathit{error}\ \texttt{"make illegal states unrepresentable"}
\end{array}
$$

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

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

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{AST} = \\
&\quad \mathit{Lit}\ \mathit{Term} \\
&\quad \mid \mathit{AddU}\ \mathit{AST}\ \mathit{AST} \\
&\quad \mid \mathit{AndU}\ \mathit{AST}\ \mathit{AST} \\
&\mathit{twoPlusTwo}' :: \mathit{AST} \\
&\mathit{twoPlusTwo}' = \mathit{AddU}\ (\mathit{Lit}\ (\mathit{TInt}\ 2))\ (\mathit{Lit}\ (\mathit{TInt}\ 2))
\end{aligned}
$$

interpreter

The syntax is a typeclass

$$
\begin{array}{l}
\textbf{class}\ \mathit{AddSym}\ \mathit{repr}\ \textbf{where} \\
\quad \mathit{int} :: \mathit{Int} \rightarrow \mathit{repr}\ \mathit{Int} \\
\quad \mathit{add} :: \mathit{repr}\ \mathit{Int} \rightarrow \mathit{repr}\ \mathit{Int} \rightarrow \mathit{repr}\ \mathit{Int}
\end{array}
$$

typed tagless

The tagless-typed-ness comes from

$$repr :: * \to *$$

typed tagless

The tagless-typed-ness comes from

$$repr :: * \to *$$

Which is why the above example won't typecheck

$$
\begin{aligned}
&\mathit{int}\ 2 :: \mathit{repr}\ \mathit{Int}\\
&\mathit{bool}\ 3 :: \mathit{repr}\ \mathit{Bool}\\
&\mathit{add} :: \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int}
\end{aligned}
$$

interpreter

The semantics are an instance of the typeclass

$$
\begin{array}{l}
\textbf{newtype}\ R\ a = R\ \{\mathit{unR} :: a\} \\
\textbf{instance}\ \mathit{AddSym}\ R\ \textbf{where} \\
\quad \mathit{int} = R \\
\quad \mathit{add}\ (R\ x)\ (R\ y) = R\ \$\ x + y
\end{array}
$$

extensible

Adding interpretations

$$
\begin{array}{l}
\textbf{newtype}\ S\ a = S\ \{\, unS :: Int \to String \,\} \\
\textbf{instance}\ AddSym\ S\ \textbf{where} \\
\quad int = S \circ const \circ show \\
\quad add\ (S\ x)\ (S\ y) = S\ \$\ \lambda c \to \\
\qquad \texttt{"("} <> x\ c <> \texttt{" + "} <> y\ c <> \texttt{")"}
\end{array}
$$

extensible

Adding operations

```
class MultSym repr where
  mul :: repr Int → repr Int → repr Int
instance MultSym R where
  mul (R x) (R y) = R $ x ∗ y
instance MultSym S where
  mul (S x) (S y) = S $ λc →
    "(" <> x c <> " * " <> y c <> ")"
```

adding booleans

$$
\begin{array}{l}
\textbf{class}\ \mathit{BoolSym}\ \mathit{repr}\ \textbf{where} \\
\quad \mathit{bool} :: \mathit{Bool} \to \mathit{repr}\ \mathit{Bool} \\
\quad \mathit{lte} :: \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Int} \to \mathit{repr}\ \mathit{Bool} \\
\quad \mathit{when} :: \mathit{repr}\ \mathit{Bool} \to \mathit{repr}\ a \to \mathit{repr}\ a \to \mathit{repr}\ a
\end{array}
$$

adding booleans

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

adding booleans

```haskell
instance BoolSym R where
  bool = R
  lte (R x) (R y) = R $ x <= y
  when (R b) (R t) (R f) = R $ if b then t else f
```

```haskell
instance BoolSym S where
  bool = S . const . show
  lte (S x) (S y) = S $ \c ->
    "(" <> x c <> " <= " <> y c <> ")"
  when (S b) (S t) (S f) = S $ \c ->
    "if " <> b c
     <> "\nthen " <> t c
     <> "\n else" <> f c
```

adding lambda abstraction

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

adding lambda abstraction

instance *LamSym R* **where**
  $lam\ f = R\ \$\ unR \circ f \circ R$
  $app\ (R\ f)\ (R\ a) = R\ \$\ f\ a$

adding lambda abstraction

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

$$
\begin{array}{l}
\textbf{instance}\ \mathit{LamSym}\ S\ \textbf{where} \\
\quad \mathit{lam}\ f\ = S\ \$\ \lambda c \to \\
\qquad \textbf{let} \\
\qquad\quad x = \texttt{"x"} <> \mathit{show}\ c \\
\qquad \textbf{in} \\
\qquad\quad \texttt{"\\\\"} <> x <> \texttt{".\\n"} \\
\qquad\quad <> (\mathit{unS} \circ f \circ S\ \$\ \mathit{const}\ x)\ (c + 1) \\
\quad \mathit{app}\ (S\ f)\ (S\ a) = S\ \$\ \lambda c \to \\
\qquad \texttt{"("} <> f\ c <> \texttt{" "} <> a\ c <> \texttt{")"}
\end{array}
$$

higher-order abstract syntax

Notice that the metalanguage (Haskell) maintains the variable bindings

adding recursion!!

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

adding recursion!!

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

adding recursion!!

```
instance FixSym R where
  fix f = R $ fx (unR ∘ f ∘ R)
    where fx g = g (fx g)
```

```
instance FixSym S where
  fix f = S $ λc →
    let
      self = "self" <> show c
    in
      "(fix " <> self <> ".\n"
       <> (unS ∘ f ∘ S $ const self) (c + 1) <> ")"
```

simply typed lambda calculus with integer and boolean literals

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

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

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

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

factorial

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

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

contrast: typed tagless initial encoding

$$
\begin{array}{l}
\textbf{data}\ \mathit{SymI}\ h\ a\ \textbf{where} \\
\quad \mathit{INT} :: \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \\
\quad \mathit{Add} :: \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \\
\quad \mathit{Mul} :: \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \\
\quad \mathit{BOOL} :: \mathit{Bool} \to \mathit{SymI}\ h\ \mathit{Bool} \\
\quad \mathit{Lte} :: \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Int} \to \mathit{SymI}\ h\ \mathit{Bool} \\
\quad \mathit{When} :: \mathit{SymI}\ h\ \mathit{Bool} \to \mathit{SymI}\ h\ a \to \mathit{SymI}\ h\ a \to \mathit{SymI}\ h\ a \\
\quad \mathit{Var} :: h\ a \to \mathit{SymI}\ h\ a \\
\quad \mathit{Lam} :: (\mathit{SymI}\ h\ a \to \mathit{SymI}\ h\ b) \to \mathit{SymI}\ h\ (a \to b) \\
\quad \mathit{App} :: \mathit{SymI}\ h\ (a \to b) \to \mathit{SymI}\ h\ a \to \mathit{SymI}\ h\ b \\
\quad \mathit{Fix} :: (\mathit{SymI}\ h\ a \to \mathit{SymI}\ h\ a) \to \mathit{SymI}\ h\ a
\end{array}
$$

`h` is the evaluation context (eg `R` or `S` etc)

Like the final encoding, this is well typed (if it compiles, it won't get stuck) and there are no "type tags".

Like the final encoding, this is well typed (if it compiles, it won't get stuck) and there are no "type tags".

This requires `GADTs` so that the result of pattern-matching on different branches of a sum type can have different types

initial interpreter

Here we evaluate in `R`

$$
\begin{array}{l}
\mathit{evalI} :: \mathit{SymI}\ R\ a \to a \\
\mathit{evalI}\ (\mathit{INT}\ i) = i \\
\mathit{evalI}\ (\mathit{Add}\ x\ y) = (\mathit{evalI}\ x) + (\mathit{evalI}\ y) \\
\mathit{evalI}\ (\mathit{Mul}\ x\ y) = (\mathit{evalI}\ x) * (\mathit{evalI}\ y) \\
\mathit{evalI}\ (\mathit{BOOL}\ b) = b \\
\mathit{evalI}\ (\mathit{Lte}\ x\ y) = (\mathit{evalI}\ x) \leqslant (\mathit{evalI}\ y) \\
\mathit{evalI}\ (\mathit{When}\ b\ t\ f) = \mathbf{if}\ (\mathit{evalI}\ b)\ \mathbf{then}\ (\mathit{evalI}\ t)\ \mathbf{else}\ (\mathit{evalI}\ f) \\
\mathit{evalI}\ (\mathit{Var}\ x) = \mathit{unR}\ x \\
\mathit{evalI}\ (\mathit{Lam}\ f) = \mathit{evalI} \circ f \circ \mathit{Var} \circ R \\
\mathit{evalI}\ (\mathit{App}\ f\ a) = (\mathit{evalI}\ f)\ (\mathit{evalI}\ a) \\
\mathit{evalI}\ (\mathit{Fix}\ f) = \mathit{evalI}\ (\mathit{fx}\ f) \\
\quad \mathbf{where}\ \mathit{fx}\ g = g\ (\mathit{fx}\ g)
\end{array}
$$

expression problem

Adding interpreters to `SymI` is straightforward

$$pprintI :: Sym\ S\ a \to String$$
$$pprintI = \ldots$$

expression problem

Adding interpreters to `SymI` is straightforward

$$\begin{aligned} &pprintI :: Sym\ S\ a \rightarrow String \\ &pprintI = \ldots \end{aligned}$$

But adding new operations isn't possible

expression problem

Adding interpreters to `SymI` is straightforward

$$pprintI :: Sym\ S\ a \rightarrow String$$
$$pprintI = \ldots$$

But adding new operations isn't possible

Because of the contravariant `SymI` in `Lam`, `SymI` is not representable as the fixpoint of a functor, and so we can't use the coproduct-of-functors representation.

bijection

We can show the final and initial embeddings are equivalent by establishing a bijection

final to initial

instance $AddSym\ (SymI\ \ h)$ **where**
  $int = INT$
  $add = Add$
instance $MultSym\ (SymI\ \ h)$ **where**
  $mul = Mul$
instance $BoolSym\ (SymI\ \ h)$ **where**
  $bool = BOOL$
  $lte = Lte$
  $when = When$
instance $LamSym\ (SymI\ \ h)$ **where**
  $lam = Lam$
  $app = App$
instance $FixSym\ (SymI\ \ h)$ **where**
  $fix = Fix$

instance $Symantics\ (SymI\ \ h)$

$fToI :: SymI\ \ h\ a \rightarrow SymI\ \ h\ a$
$fToI = id$

initial to final

$$
\begin{aligned}
&iToF :: (Symantics\ repr) \Rightarrow SymI\ repr\ a \rightarrow repr\ a\\
&iToF\ (INT\ x) = int\ x\\
&iToF\ (Add\ x\ y) = add\ (iToF\ x)\ (iToF\ y)\\
&iToF\ (Mul\ x\ y) = mul\ (iToF\ x)\ (iToF\ y)\\
&iToF\ (BOOL\ b) = bool\ b\\
&iToF\ (Lte\ x\ y) = lte\ (iToF\ x)\ (iToF\ y)\\
&iToF\ (When\ b\ t\ f) = when\ (iToF\ b)\ (iToF\ t)\ (iToF\ f)\\
&iToF\ (Var\ a) = a\\
&iToF\ (Lam\ f) = lam\ (\lambda x \rightarrow iToF\ (f\ (Var\ x)))\\
&iToF\ (App\ f\ a) = app\ (iToF\ f)\ (iToF\ a)\\
&iToF\ (Fix\ f) = fix\ (\lambda self \rightarrow iToF\ (f\ (Var\ self)))
\end{aligned}
$$

end

Check out Oleg's writeups at

- http://okmij.org/ftp/tagless-final/index.html
- http://okmij.org/ftp/tagless-final/course/index.html

This presentation and some code snippets at https://github.com/shterrett/ttfi