An
Algebraic Approach
to
Typechecking and Elaboration

Robert Atkey
bob.atkey@ed.ac.uk

Wednesday 18th February 2015

Let's write a typechecker

Let's write a typechecker

Let's write a typechecker

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

Let's write a typechecker

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{data}\ \mathit{Term} = \mathsf{Var}\ \mathit{Int} \mid \mathsf{Lam}\ \mathit{Type}\ \mathit{Term} \mid \mathsf{App}\ \mathit{Term}\ \mathit{Term}$$

Let's write a typechecker

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

data *Term* = Var *Int* | Lam *Type Term* | App *Term Term*

typecheck :: *Term* → [*Type*] → *Maybe Type*

Let's write a typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

typecheck :: Term → [Type] → Maybe Type
typecheck (Var i) ctxt          = Just (ctxt !! i)
```

Let's write a typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

typecheck :: Term → [Type] → Maybe Type
typecheck (Var i) ctxt          = Just (ctxt !! i)
typecheck (Lam ty tm) ctxt      = case typecheck tm (ty:ctxt) of
                                    Just ty′ → Just (ty ⇒ ty′)
                                    Nothing → Nothing
```

Let's write a typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

typecheck :: Term → [Type] → Maybe Type
typecheck (Var i) ctxt          = Just (ctxt !! i)
typecheck (Lam ty tm) ctxt      = case typecheck tm (ty:ctxt) of
                                    Just ty' → Just (ty ⇒ ty')
                                    Nothing → Nothing
typecheck (App tm1 tm2) ctxt = case typecheck tm1 ctxt of
                                    Just (ty1 ⇒ ty2) →
                                      case typecheck tm2 ctxt of
                                        Just ty1' | ty1 ≡ ty1' → Just ty2
                                        _ → Nothing
                                    _ → Nothing
```

Let's take a typechecker to bits

Let's take a Typechecker to bits

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{data}\ \mathit{Term} = \mathsf{Var}\ \mathit{Int} \mid \mathsf{Lam}\ \mathit{Type}\ \mathit{Term} \mid \mathsf{App}\ \mathit{Term}\ \mathit{Term}$$

Let's take a Typechecker to bits

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{data}\ \mathit{Term} = \mathsf{Var}\ \mathit{Int} \mid \mathsf{Lam}\ \mathit{Type}\ \mathit{Term} \mid \mathsf{App}\ \mathit{Term}\ \mathit{Term}$$

$$\textbf{type}\ \mathit{TypeChecker} = [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type}$$

Let's take a Typechecker to bits

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{data}\ \mathit{Term} = \mathsf{Var}\ \mathit{Int} \mid \mathsf{Lam}\ \mathit{Type}\ \mathit{Term} \mid \mathsf{App}\ \mathit{Term}\ \mathit{Term}$$

$$\textbf{type}\ \mathit{TypeChecker} = [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type}$$

$$\mathrm{typecheck} :: \mathit{Term} \to \mathit{TypeChecker}$$

Let's take a Typechecker to bits

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

type TypeChecker = [Type] → Maybe Type

typecheck :: Term → TypeChecker
typecheck (Var i)          = var i
```

Let's take a Typechecker to bits

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

type TypeChecker = [Type] → Maybe Type

typecheck :: Term → TypeChecker
typecheck (Var i)          = var i
typecheck (Lam ty tm)      = lam ty (typecheck tm)
```

Let's take a Typechecker to bits

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

data Term = Var Int | Lam Type Term | App Term Term

type TypeChecker = [Type] → Maybe Type

typecheck :: Term → TypeChecker
typecheck (Var i)          = var i
typecheck (Lam ty tm)      = lam ty (typecheck tm)
typecheck (App tm1 tm2)    = app (typecheck tm1) (typecheck tm2)
```

Bits of a Typechecker

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{type}\ \mathit{TypeChecker} = [\mathit{Type}] \rightarrow \mathit{Maybe}\ \mathit{Type}$$

Bits of a Typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

type TypeChecker = [Type] → Maybe Type

var :: Int → TypeChecker
var i = λctxt → Just (ctxt !! i)
```

Bits of a Typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

type TypeChecker = [Type] → Maybe Type

var :: Int → TypeChecker
var i = λctxt → Just (ctxt !! i)

lam :: Type → TypeChecker → TypeChecker
lam ty tc = case typecheck tm (ty:ctxt) of
                Just ty′ → Just (ty ⇒ ty′);    Nothing → Nothing
```

Bits of a Typechecker

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

type TypeChecker = [Type] → Maybe Type

var :: Int → TypeChecker
var i = λctxt → Just (ctxt !! i)

lam :: Type → TypeChecker → TypeChecker
lam ty tc = case typecheck tm (ty:ctxt) of
                Just ty′ → Just (ty ⇒ ty′);    Nothing → Nothing

app :: TypeChecker → TypeChecker → TypeChecker
app tc₁ tc₂ = λctxt → case tc₁ ctxt of
                        Just (ty₁ ⇒ ty₂) →
                          case tc₂ ctxt of
                            Just ty₁′ | ty₁ ≡ ty₁′ → Just ty₂
                            _ → Nothing
                        _ → Nothing
```

Typechecker Scripts

With these bits, we can write *typechecker scripts*.

A term: “$\lambda f{:}A \Rightarrow B.\ \lambda a{:}A.\ f\,a$”
& its typechecker: $\mathsf{lam}\ (\mathsf{A} \Rightarrow \mathsf{B})\ (\mathsf{lam}\ \mathsf{A}\ (\mathsf{app}\ (\mathsf{var}\ 1)\ (\mathsf{var}\ 0)))$

A term family: “$\lambda f{:}A \Rightarrow B.\ \lambda a{:}A.\ [-]$”
& its typechecker: $\lambda x.\ \mathsf{lam}\ (\mathsf{A} \Rightarrow \mathsf{B})\ (\mathsf{lam}\ \mathsf{A}\ x)$

More Typechecker Bits

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\textbf{type}\ \mathit{TypeChecker} = [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type}$$

More Typechecker Bits

```
data Type = A | B | C | Type => Type deriving (Eq)

type TypeChecker = [Type] -> Maybe Type

failure :: TypeChecker
failure = λctxt -> Nothing
```

More Typechecker Bits

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

type TypeChecker = [Type] → Maybe Type

failure :: TypeChecker
failure = λctxt → Nothing

have :: Int → Type → TypeChecker → TypeChecker
have i ty tc = λctxt → if ctxt !! i ≡ ty then tc ctxt else Nothing
```

More Typechecker Bits

```
data Type = A | B | C | Type ⇒ Type deriving (Eq)

type TypeChecker = [Type] → Maybe Type

failure :: TypeChecker
failure = λctxt → Nothing

have :: Int → Type → TypeChecker → TypeChecker
have i ty tc = λctxt → if ctxt !! i ≡ ty then tc ctxt else Nothing

hasType :: Type → TypeChecker → TypeChecker
hasType ty tc = λctxt → case tc ctxt of
                          Just ty′ | ty ≡ ty′ → Just ty
                          _ → Nothing
```

Typechecker Scripts

A term, with an assertion

$$\mathsf{hasType}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})$$
$$(\mathrm{lam}\ (\mathsf{A} \Rightarrow \mathsf{B})\ (\mathrm{lam}\ \mathsf{A}\ (\mathrm{app}\ (\mathrm{var}\ 1)\ (\mathrm{var}\ 0))))$$

Typechecker Scripts

A term, with an assertion

$$\begin{aligned}&\mathsf{hasType}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})\\ &\qquad (\mathsf{lam}\ (\mathsf{A} \Rightarrow \mathsf{B})\ (\mathsf{lam}\ \mathsf{A}\ (\mathsf{app}\ (\mathsf{var}\ 1)\ (\mathsf{var}\ 0))))\end{aligned}$$

A term with a hole, with an assertion

$$\begin{aligned}&\lambda \mathrm{x}.\ \mathsf{hasType}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})\\ &\qquad (\mathsf{lam}\ (\mathsf{A} \Rightarrow \mathsf{B})\ (\mathsf{lam}\ \mathsf{A}\ \mathrm{x}))\end{aligned}$$

Typechecker Scripts

A term, with an assertion

$$\text{hasType}\ ((A \Rightarrow B) \Rightarrow A \Rightarrow B)\ (\text{lam}\ (A \Rightarrow B)\ (\text{lam}\ A\ (\text{app}\ (\text{var}\ 1)\ (\text{var}\ 0))))$$

A term with a hole, with an assertion

$$\lambda x.\ \text{hasType}\ ((A \Rightarrow B) \Rightarrow A \Rightarrow B)\ (\text{lam}\ (A \Rightarrow B)\ (\text{lam}\ A\ x))$$

A term with a hole, with two assertions

$$\lambda x.\ \text{hasType}\ ((A \Rightarrow B) \Rightarrow A \Rightarrow B)\ (\text{lam}\ (A \Rightarrow B)\ (\text{lam}\ A\ (\text{have}\ 1\ (A \Rightarrow B)\ x))$$

Algebraic Theory of Typechecking

The Algebraic Theory of Typechecking

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

class TypeChecker α **where**

var	:: $Int \to \alpha$;	have	:: $Int \to Type \to \alpha \to \alpha$
lam	:: $Type \to \alpha \to \alpha$;	hasType	:: $Type \to \alpha \to \alpha$
app	:: $\alpha \to \alpha \to \alpha$;	failure	:: α

The Algebraic Theory of Typechecking

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

class TypeChecker α **where**

var	:: $Int \to \alpha$;	have	:: $Int \to Type \to \alpha \to \alpha$
lam	:: $Type \to \alpha \to \alpha$;	hasType	:: $Type \to \alpha \to \alpha$
app	:: $\alpha \to \alpha \to \alpha$;	failure	:: α

Equations 1: *Failure is contagious*

$$
\begin{aligned}
\text{failure} &= \text{lam } A \text{ failure} \\
&= \text{app failure } x \\
&= \text{app } x \text{ failure} \\
&= \text{have } i \ A \text{ failure} \\
&= \text{hasType } A \text{ failure}
\end{aligned}
$$

The Algebraic Theory of Typechecking

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

class TypeChecker α **where**

var	:: $Int \to \alpha$;	have	:: $Int \to Type \to \alpha \to \alpha$
lam	:: $Type \to \alpha \to \alpha$;	hasType	:: $Type \to \alpha \to \alpha$
app	:: $\alpha \to \alpha \to \alpha$;	failure	:: α

Equations 2: *To have and to have not*

$$
\begin{aligned}
\text{lam } A\ x &= \text{lam } A\ (\text{have } 0\ A\ x) \\
\text{have } n\ A\ (\text{lam } B\ x) &= \text{lam } B\ (\text{have } (n+1)\ A\ x) \\
\text{have } n\ A\ (\text{app } x\ y) &= \text{app } (\text{have } n\ A\ x)\ (\text{have } n\ A\ y) \\
\text{have } n\ A\ (\text{var } n) &= \text{hasType } A\ (\text{var } n)
\end{aligned}
$$

$$
\begin{aligned}
\text{have } n\ A\ (\text{have } n\ A\ x) &= \text{have } n\ A\ x \\
\text{have } n\ A\ (\text{have } n\ B\ x) &= \text{failure} \qquad (A \neq B) \\
\text{have } n\ A\ (\text{have } n'\ B\ x) &= \text{have } n'\ B\ (\text{have } n\ A\ x) \qquad (n \neq n')
\end{aligned}
$$

The Algebraic Theory of Typechecking

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

class TypeChecker α **where**

var	:: $Int \to \alpha$;	have	:: $Int \to Type \to \alpha \to \alpha$
lam	:: $Type \to \alpha \to \alpha$;	hasType	:: $Type \to \alpha \to \alpha$
app	:: $\alpha \to \alpha \to \alpha$;	failure	:: α

Equations 3: *I can hasType*

$$
\begin{aligned}
&\text{lam } A \text{ (hasType } B\ x) && = \text{ hasType } (A \Rightarrow B) \text{ (lam } A\ x) \\
&\text{app (hasType } (A \Rightarrow B)\ x) && = \text{ hasType } B \text{ (app (hasType } (A \Rightarrow B)\ x)\ y) \\
&\quad \text{(hasType } A\ y) \\
&\text{app (hasType } A\ x)\ y && = \text{ failure} \qquad (A \neq - \Rightarrow -) \\
&\text{app (hasType } (A \Rightarrow B)\ x) && = \text{ failure} \qquad (A \neq C) \\
&\quad \text{(hasType } C\ y) \\
&\text{hasType } A \text{ (hasType } A\ x) && = \text{ hasType } A\ x \\
&\text{hasType } A \text{ (hasType } B\ x) && = \text{ failure} \qquad (A \neq B)
\end{aligned}
$$

Equations vs. Actual Typing

For any term t, let $[\![t]\!]$ be the translation into the Typechecker theory using var, lam, and app.

Theorem

$$\vdash t : A$$

if and only if

$$\text{hasType } A\ [\![t]\!] \neq \text{failure}$$

(in the Typechecker theory)

Have an Algebraic Theory? Think "Monad!"

$$
\begin{aligned}
\textbf{data}\ \mathit{TCTerm}\ \alpha = {} & \mathsf{Return} && \alpha \\
\mid {} & \mathsf{Var} && \mathit{Int} \\
\mid {} & \mathsf{Lam} && \mathit{Type}\ (\mathit{TCTerm}\ \alpha) \\
\mid {} & \mathsf{App} && (\mathit{TCTerm}\ \alpha)\ (\mathit{TCTerm}\ \alpha) \\
\mid {} & \mathsf{Have} && \mathit{Int}\ \mathit{Type}\ (\mathit{TCTerm}\ \alpha) \\
\mid {} & \mathsf{HasType} && \mathit{Type}\ (\mathit{TCTerm}\ \alpha) \\
\mid {} & \mathsf{Failure}
\end{aligned}
$$

$$
\begin{aligned}
&(\gg\!=) :: \mathit{TCTerm}\ \alpha \to (\alpha \to \mathit{TCTerm}\ \beta) \to \mathit{TCTerm}\ \beta \\
&\mathsf{Return\ a} && \gg\!= \mathsf{f} = \mathsf{f\ a} \\
&\mathsf{Var\ i} && \gg\!= \mathsf{f} = \mathsf{Var\ i} \\
&\mathsf{Lam\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Lam\ ty}\ (\mathsf{t} \gg\!= \mathsf{f}) \\
&\mathsf{App\ t_1\ t_2} && \gg\!= \mathsf{f} = \mathsf{App}\ (\mathsf{t_1} \gg\!= \mathsf{f})\ (\mathsf{t_2} \gg\!= \mathsf{f}) \\
&\mathsf{Have\ i\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Have\ i\ ty}\ (\mathsf{t} \gg\!= \mathsf{f}) \\
&\mathsf{HasType\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{HasType\ ty}\ (\mathsf{t} \gg\!= \mathsf{f}) \\
&\mathsf{Failure} && \gg\!= \mathsf{f} = \mathsf{Failure}
\end{aligned}
$$

Typechecker Scripts

For, any α, *TCTerm* α is a free Typechecker algebra
(rules shall be imposed later)

Some algebraic operations, and generic effects:

$\mathsf{var}\ i = \mathsf{Var}\ i$
$\mathsf{lam}\ A = \mathsf{Lam}\ A\ (\mathsf{Return}\ ())$
$\mathsf{app}\ x\ y = \mathsf{App}\ x\ y$
$\mathsf{have}\ n\ A = \mathsf{Have}\ n\ A\ (\mathsf{Return}\ ())$
$\mathsf{goalIs}\ A = \mathsf{HasType}\ A\ (\mathsf{Return}\ ())$
$\mathsf{failure} = \mathsf{Failure}$

A typechecker script, monadic style:

do goalIs $((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})$
lam $(\mathsf{A} \Rightarrow \mathsf{B})$
lam A
have 1 $(\mathsf{A} \Rightarrow \mathsf{B})$
have 0 A
goalIs B
app (var 1) (var 0)

Sorting out Scoping, Method A: de Bruijn

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ (\alpha :: \mathit{Nat} \to *)\ \textbf{where}\\
&\begin{array}{llll}
\mathrm{var} & :: \mathit{Fin}\, n \to \alpha\, n; & \mathrm{have} & :: \mathit{Fin}\, n \to \mathit{Type} \to \alpha\, n \to \alpha\, n\\
\mathrm{lam} & :: \mathit{Type} \to \alpha\, (\mathrm{Suc}\, n) \to \alpha\, n; & \mathrm{hasType} & :: \mathit{Type} \to \alpha\, n \to \alpha\, n\\
\mathrm{app} & :: \alpha\, n \to \alpha\, n \to \alpha\, n; & \mathrm{failure} & :: \alpha\, n
\end{array}
\end{aligned}$$

(Abstract Syntax and Variable Binding, Fiore, Plotkin, Turi, LICS 1999)

So $\alpha\ n$ is a typechecker in a context with n free variables

Same equations...

Sorting out Scoping, Method B: HOAS

data *Type* = A | B | C | *Type* ⇒ *Type* **deriving** (Eq)

class TypeChecker ν α **where**

var	:: $\nu \to \alpha$;	have	:: $\nu \to \mathit{Type} \to \alpha \to \alpha$
lam	:: $\mathit{Type} \to (\nu \to \alpha) \to \alpha$;	hasType	:: $\mathit{Type} \to \alpha \to \alpha$
app	:: $\alpha \to \alpha \to \alpha$;	failure	:: α

The abstract type ν represents variables.

Using a lam gets us a new variable.

Typecheckers with no free variables are represented by the type:

$$\forall \nu\ \alpha.\ \text{TypeChecker}\ \nu\ \alpha \Rightarrow \alpha$$

(Equivalent to previous (Syntax for free..., Atkey, TLCA 2009))

Equations???

Have an Algebraic Theory? Think "Monad!"

$$
\begin{aligned}
\textbf{data}\ \mathit{TCTerm}\ \nu\ \alpha = {} & \mathsf{Return} && \alpha \\
\mid {} & \mathsf{Var} && \nu \\
\mid {} & \mathsf{Lam} && \mathit{Type}\ (\nu \to \mathit{TCTerm}\ \nu\ \alpha) \\
\mid {} & \mathsf{App} && (\mathit{TCTerm}\ \nu\ \alpha)\ \ (\mathit{TCTerm}\ \nu\ \alpha) \\
\mid {} & \mathsf{Have} && \nu\ \ \mathit{Type}\ \ (\mathit{TCTerm}\ \nu\ \alpha) \\
\mid {} & \mathsf{HasType} && \mathit{Type}\ \ (\mathit{TCTerm}\ \nu\ \alpha) \\
\mid {} & \mathsf{Failure}
\end{aligned}
$$

$$
\begin{aligned}
&(\gg\!=) :: \mathit{TCTerm}\ \nu\ \alpha \to (\alpha \to \mathit{TCTerm}\ \nu\ \beta) \to \mathit{TCTerm}\ \nu\ \beta \\
&\mathsf{Return\ a} && \gg\!= \mathsf{f} = \mathsf{f\ a} \\
&\mathsf{Var\ v} && \gg\!= \mathsf{f} = \mathsf{Var\ v} \\
&\mathsf{Lam\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Lam\ ty}\ \ (\lambda \mathsf{v.\ t\ v} \gg\!= \mathsf{f}) \\
&\mathsf{App\ t_1\ t_2} && \gg\!= \mathsf{f} = \mathsf{App}\ \ (\mathsf{t_1} \gg\!= \mathsf{f})\ \ (\mathsf{t_2} \gg\!= \mathsf{f}) \\
&\mathsf{Have\ v\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Have\ v\ ty}\ \ (\mathsf{t} \gg\!= \mathsf{f}) \\
&\mathsf{HasType\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{HasType\ ty}\ \ (\mathsf{t} \gg\!= \mathsf{f}) \\
&\mathsf{Failure} && \gg\!= \mathsf{f} = \mathsf{Failure}
\end{aligned}
$$

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

$$\begin{aligned}
&\mathsf{var}\ \mathsf{v} = \mathsf{Var}\ \mathsf{v} \\
&\mathsf{lam}\ A = \mathsf{Lam}\ A\ (\lambda \mathsf{v}.\ \mathsf{Return}\ \mathsf{v}) \\
&\mathsf{app}\ x\ y = \mathsf{App}\ x\ y \\
&\mathsf{have}\ \mathsf{v}\ A = \mathsf{Have}\ \mathsf{v}\ A\ (\mathsf{Return}\ ()) \\
&\mathsf{goalIs}\ A = \mathsf{HasType}\ A\ (\mathsf{Return}\ ()) \\
&\mathsf{failure} = \mathsf{Failure}
\end{aligned}$$

A typechecker script, HOAS monadic style:

$$\begin{aligned}
&\textbf{do}\ \mathsf{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B}) \\
&\quad \mathsf{v}_1 \leftarrow \mathsf{lam}\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
&\quad \mathsf{v}_2 \leftarrow \mathsf{lam}\ \mathsf{A} \\
&\quad \mathsf{have}\ \mathsf{v}_1\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
&\quad \mathsf{have}\ \mathsf{v}_2\ \mathsf{A} \\
&\quad \mathsf{goalIs}\ \mathsf{B} \\
&\quad \mathsf{app}\ (\mathsf{var}\ \mathsf{v}_1)\ (\mathsf{var}\ \mathsf{v}_2)
\end{aligned}$$

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

$$\mathsf{var}\ \mathrm{v} = \mathsf{Var}\ \mathrm{v}$$
$$\mathsf{introduce}\ A = \mathsf{Lam}\ A\ (\lambda \mathrm{v}.\ \mathsf{Return}\ \mathrm{v})$$
$$\mathsf{app}\ x\ y = \mathsf{App}\ x\ y$$
$$\mathsf{have}\ \mathrm{v}\ A = \mathsf{Have}\ \mathrm{v}\ A\ (\mathsf{Return}\ ())$$
$$\mathsf{goalIs}\ A = \mathsf{HasType}\ A\ (\mathsf{Return}\ ())$$
$$\mathsf{failure} = \mathsf{Failure}$$

A typechecker script, HOAS monadic style:

do $\mathsf{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})$
$\mathrm{v}_1 \leftarrow \mathsf{introduce}\ (\mathsf{A} \Rightarrow \mathsf{B})$
$\mathrm{v}_2 \leftarrow \mathsf{introduce}\ \mathsf{A}$
$\mathsf{have}\ \mathrm{v}_1\ (\mathsf{A} \Rightarrow \mathsf{B})$
$\mathsf{have}\ \mathrm{v}_2\ \mathsf{A}$
$\mathsf{goalIs}\ \mathsf{B}$
$\mathsf{app}\ (\mathsf{var}\ \mathrm{v}_1)\ (\mathsf{var}\ \mathrm{v}_2)$

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

$$\text{assumption}\ \mathsf{v} = \mathsf{Var}\ \mathsf{v}$$
$$\text{introduce}\ A = \mathsf{Lam}\ A\ (\lambda \mathsf{v}.\ \mathsf{Return}\ \mathsf{v})$$
$$\mathsf{app}\ x\ y = \mathsf{App}\ x\ y$$
$$\mathsf{have}\ \mathsf{v}\ A = \mathsf{Have}\ \mathsf{v}\ A\ (\mathsf{Return}\ ())$$
$$\mathsf{goalIs}\ A = \mathsf{HasType}\ A\ (\mathsf{Return}\ ())$$
$$\mathsf{failure} = \mathsf{Failure}$$

A typechecker script, HOAS monadic style:

do $\mathsf{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})$
$\mathsf{v}_1 \leftarrow \text{introduce}\ (\mathsf{A} \Rightarrow \mathsf{B})$
$\mathsf{v}_2 \leftarrow \text{introduce}\ \mathsf{A}$
$\mathsf{have}\ \mathsf{v}_1\ (\mathsf{A} \Rightarrow \mathsf{B})$
$\mathsf{have}\ \mathsf{v}_2\ \mathsf{A}$
$\mathsf{goalIs}\ \mathsf{B}$
$\mathsf{app}\ (\text{assumption}\ \mathsf{v}_1)\ (\text{assumption}\ \mathsf{v}_2)$

Evaluating Typechecker Scripts

$$
\begin{array}{lcl}
\text{eval} :: \text{TypeChecker}\ \alpha \Rightarrow \mathit{TCTerm}\ \mathbf{0} \to \alpha & & \\
\text{eval}\ (\text{Return}\ ()) & & \\
\text{eval}\ (\text{Var}\ \text{i}) & = & \text{var}\ \text{i} \\
\text{eval}\ (\text{Lam}\ A\ \text{t}) & = & \text{lam}\ A\ (\text{eval}\ \text{t}) \\
\text{eval}\ (\text{App}\ \text{t}_1\ \text{t}_2) & = & \text{app}\ (\text{eval}\ \text{t}_1)\ (\text{eval}\ \text{t}_2) \\
\text{eval}\ (\text{Have}\ \text{i}\ A\ \text{t}) & = & \text{have}\ \text{i}\ A\ (\text{eval}\ t) \\
\text{eval}\ (\text{HasType}\ A\ \text{t}) & = & \text{hasType}\ A\ (\text{eval}\ t) \\
\text{eval}\ \text{Failure} & = & \text{failure}
\end{array}
$$

Evaluating Typechecker Scripts

$$
\begin{array}{lcl}
\text{eval} :: \text{TypeChecker}\ \alpha \Rightarrow \mathit{TCTerm}\ \mathbf{0} \to \alpha & & \\
\text{eval}\ (\text{Return}\ ()) & & \\
\text{eval}\ (\text{Var}\ \text{i}) & = & \text{var}\ \text{i} \\
\text{eval}\ (\text{Lam}\ A\ \text{t}) & = & \text{lam}\ A\ (\text{eval}\ \text{t}) \\
\text{eval}\ (\text{App}\ \text{t}_1\ \text{t}_2) & = & \text{app}\ (\text{eval}\ \text{t}_1)\ (\text{eval}\ \text{t}_2) \\
\text{eval}\ (\text{Have}\ \text{i}\ A\ \text{t}) & = & \text{have}\ \text{i}\ A\ (\text{eval}\ t) \\
\text{eval}\ (\text{HasType}\ A\ \text{t}) & = & \text{hasType}\ A\ (\text{eval}\ t) \\
\text{eval}\ \text{Failure} & = & \text{failure}
\end{array}
$$

Type checking:

instance TypeChecker ([*Type*] → *Maybe Type*) **where**...

Evaluating Typechecker Scripts

$$
\begin{aligned}
&\text{eval} :: \text{TypeChecker}\ \alpha \Rightarrow \mathit{TCTerm}\ \mathbf{0} \to \alpha \\
&\text{eval}\ (\text{Return}\ ()) \\
&\text{eval}\ (\text{Var i}) &&= \text{var i} \\
&\text{eval}\ (\text{Lam}\ A\ \text{t}) &&= \text{lam}\ A\ (\text{eval t}) \\
&\text{eval}\ (\text{App}\ \text{t}_1\ \text{t}_2) &&= \text{app}\ (\text{eval}\ \text{t}_1)\ (\text{eval}\ \text{t}_2) \\
&\text{eval}\ (\text{Have i}\ A\ \text{t}) &&= \text{have i}\ A\ (\text{eval}\ t) \\
&\text{eval}\ (\text{HasType}\ A\ \text{t}) &&= \text{hasType}\ A\ (\text{eval}\ t) \\
&\text{eval Failure} &&= \text{failure}
\end{aligned}
$$

Type checking:

instance TypeChecker $([\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type})$ **where**...

Elaboration:

instance TypeChecker

$((\Gamma :: [\mathit{Type}]) \to \mathit{Maybe}\ ((A :: \mathit{Type}) \times \text{Tm}\ \Gamma\ A)$ **where**...

Two Useful Variations

Bidirectional Checking and Synthesis

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where} \\
&\quad \mathrm{var} &&:: \nu \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{lam} &&:: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{app} &&:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}
\end{aligned}$$

Bidirectional Checking and Synthesis

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where} \\
&\quad \mathrm{var} &&:: \nu \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{lam} &&:: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{app} &&:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{switch} &&:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{ascribe} &&:: \mathit{Type} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}
\end{aligned}$$

Bidirectional Checking and Synthesis

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where} \\
&\quad \mathrm{var} && :: \nu \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{lam} && :: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{app} && :: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{switch} && :: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{ascribe} && :: \mathit{Type} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN} \\
&\quad \mathrm{have} && :: \nu \to (\mathit{Type} \to \alpha\ \delta) \to \alpha\ \delta \\
&\quad \mathrm{goalIs} && :: (\mathit{Type} \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
&\quad \mathrm{failure} && :: \alpha\ \delta
\end{aligned}$$

Bidirectional Checking and Synthesis

$$\textbf{data}\ \mathit{Type} = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type} \Rightarrow \mathit{Type}\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where}\\
&\quad \mathrm{var} &&:: \nu \to \alpha\ \mathsf{SYN}\\
&\quad \mathrm{lam} &&:: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK}\\
&\quad \mathrm{app} &&:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}\\
&\quad \mathrm{switch} &&:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK}\\
&\quad \mathrm{ascribe} &&:: \mathit{Type} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}\\
&\quad \mathrm{have} &&:: \nu \to (\mathit{Type} \to \alpha\ \delta) \to \alpha\ \delta\\
&\quad \mathrm{goalIs} &&:: (\mathit{Type} \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK}\\
&\quad \mathrm{failure} &&:: \alpha\ \delta
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathrm{TypeChecker}\\
&\qquad (\mathsf{CHK} \mapsto [\mathit{Type}] \to \mathit{Type} \to \mathit{Bool}; \mathsf{SYN} \mapsto [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type})\\
&\quad \textbf{where}\ldots
\end{aligned}$$

Bidirectional Typechecker Scripts

Terms that are "active" in their environment:

```
do v₁ ← introduce
   v₂ ← introduce
   ty ← goal
   case ty of
     A → someAConstant
     B → someBConstant
     _ → assumption v₁
```

Bidirectional Typechecker Scripts

Terms that are "active" in their environment:

```
do v1 ← introduce
   v2 ← introduce
   ty ← goal
   case ty of
     A → someAConstant
     B → someBConstant
     _ → assumption v1
```

Application: Polymorphic Constants (elaboration from source):

```
elaborate (PolyConstant str) =
  do ty ← goal
     case ty of
       A → interpretAsAConstant str
       B → interpretAsBConstant str
       _ → failure
```

(Safely Composable Type-Specific Languages, Omar *et al.*, ECOOP 2014)

Type Inference (in Prolog)

(An Algebraic Presentation of Predicate Logic, Staton, FoSSaCS 2013)

data *Type* μ = A | B | C | *Type* $\mu \Rightarrow$ *Type* μ | MV μ **deriving** (Eq)

class TypeChecker ν μ α **where**

var :: $\nu \to \alpha$

lam :: $(\nu \to \alpha) \to \alpha$

app :: $\alpha \to \alpha \to \alpha$

Type Inference (in Prolog)

(An Algebraic Presentation of Predicate Logic, Staton, FoSSaCS 2013)

$$\textbf{data}\ \mathit{Type}\ \mu = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type}\ \mu \Rightarrow \mathit{Type}\ \mu \mid \mathsf{MV}\ \mu\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \mu\ \alpha\ \textbf{where}\\
&\quad \mathrm{var} \quad :: \nu \to \alpha\\
&\quad \mathrm{lam} \quad :: (\nu \to \alpha) \to \alpha\\
&\quad \mathrm{app} \quad :: \alpha \to \alpha \to \alpha\\
&\quad \mathrm{newMVar} \quad :: (\mu \to \alpha) \to \alpha\\
&\quad \mathrm{unify} \quad :: \mathit{Type}\ \mu \to \mathit{Type}\ \mu \to \alpha \to \alpha
\end{aligned}$$

Type Inference (in Prolog)

(An Algebraic Presentation of Predicate Logic, Staton, FoSSaCS 2013)

$$\textbf{data}\ \mathit{Type}\ \mu = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type}\ \mu \Rightarrow \mathit{Type}\ \mu \mid \mathsf{MV}\ \mu\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \mu\ \alpha\ \textbf{where}\\
&\quad \mathrm{var} \quad :: \nu \to \alpha\\
&\quad \mathrm{lam} \quad :: (\nu \to \alpha) \to \alpha\\
&\quad \mathrm{app} \quad :: \alpha \to \alpha \to \alpha\\
&\quad \mathrm{newMVar} \quad :: (\mu \to \alpha) \to \alpha\\
&\quad \mathrm{unify} \quad :: \mathit{Type}\ \mu \to \mathit{Type}\ \mu \to \alpha \to \alpha\\
&\quad \mathrm{have} \quad :: ([(\nu, \mathit{Type}\ \mu)] \to \alpha) \to \alpha\\
&\quad \mathrm{goalIs} \quad :: (\mathit{Type}\ \mu \to \alpha) \to \alpha
\end{aligned}$$

Type Inference (in Prolog)

(An Algebraic Presentation of Predicate Logic, Staton, FoSSaCS 2013)

data $\mathit{Type}\ \mu = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type}\ \mu \Rightarrow \mathit{Type}\ \mu \mid \mathsf{MV}\ \mu$ **deriving** (Eq)

class TypeChecker $\nu\ \mu\ \alpha$ **where**

var	$:: \nu \to \alpha$
lam	$:: (\nu \to \alpha) \to \alpha$
app	$:: \alpha \to \alpha \to \alpha$
newMVar	$:: (\mu \to \alpha) \to \alpha$
unify	$:: \mathit{Type}\ \mu \to \mathit{Type}\ \mu \to \alpha \to \alpha$
have	$:: ([(\nu, \mathit{Type}\ \mu)] \to \alpha) \to \alpha$
goalIs	$:: (\mathit{Type}\ \mu \to \alpha) \to \alpha$
failure	$:: \alpha$
choice	$:: \alpha \to \alpha \to \alpha$

Type Inference (in Prolog)

(An Algebraic Presentation of Predicate Logic, Staton, FoSSaCS 2013)

$$\textbf{data}\ \mathit{Type}\ \mu = \mathsf{A} \mid \mathsf{B} \mid \mathsf{C} \mid \mathit{Type}\ \mu \Rightarrow \mathit{Type}\ \mu \mid \mathsf{MV}\ \mu\ \textbf{deriving}\ (\mathrm{Eq})$$

$$\begin{aligned}
&\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \mu\ \alpha\ \textbf{where}\\
&\quad \mathrm{var} &&:: \nu \to \alpha\\
&\quad \mathrm{lam} &&:: (\nu \to \alpha) \to \alpha\\
&\quad \mathrm{app} &&:: \alpha \to \alpha \to \alpha\\
&\quad \mathrm{newMVar} &&:: (\mu \to \alpha) \to \alpha\\
&\quad \mathrm{unify} &&:: \mathit{Type}\ \mu \to \mathit{Type}\ \mu \to \alpha \to \alpha\\
&\quad \mathrm{have} &&:: ([(\nu, \mathit{Type}\ \mu)] \to \alpha) \to \alpha\\
&\quad \mathrm{goalIs} &&:: (\mathit{Type}\ \mu \to \alpha) \to \alpha\\
&\quad \mathrm{failure} &&:: \alpha\\
&\quad \mathrm{choice} &&:: \alpha \to \alpha \to \alpha
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathrm{TypeChecker}\ (\mathit{Int} \to \mathit{Int})\ \mathit{MV}\\
&\qquad (\mathit{MetaContext} \to [\mathit{Type}\ \mathit{MV}] \to \mathit{Type}\ \mathit{MV} \to \mathit{Bool})\ \textbf{where}\ldots
\end{aligned}$$

Typechecker Scripts with Unification

A "by assumption" tactic for programming:

```
byAssumption =
  do g ← goal
     ctxt ← getContext
     let search []                = failure
         search ((v, ty):ctxt = choice (do unify ty g; return v)
                                       (search ctxt)
     v ← search ctxt
     var v
```

Summary, and Some Questions

Summary:

1. Treat the bits of a typechecker as operations
2. Typechecker scripts
3. Elaboration implementation yields well-typed terms
4. Separation of core typechecking from elaboration
5. Monadic typechecker terms allow for “active terms”

Questions:

- Does this work for more complex type systems?
- What are the right operations and equations, in general?
- What are the free algebras?
- Is this a sensible way to implement a typed language?
- What is relationship to tactic scripts? HiProofs? Isar mode?
- Does this subsume (hygenic) macros?