An Algebraic Approach to Typechecking and Elaboration

Robert Atkey
robert.atkey@strath.ac.uk

Tuesday 12th May 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

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

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

$$\mathrm{typecheck} :: \mathit{Term} \to [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type}$$

Let's write a typechecker

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

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

$$\begin{array}{ll} \mathrm{typecheck} :: \mathit{Term} \to [\mathit{Type}] \to \mathit{Maybe}\ \mathit{Type} & \\ \mathrm{typecheck}\ (\mathsf{Var}\ \mathrm{i})\ \mathrm{ctxt} & = \mathsf{Just}\ (\mathrm{ctxt}\ !!\ \mathrm{i}) \end{array}$$

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

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

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{typecheck} :: \mathit{Term} \rightarrow [\mathit{Type}] \rightarrow \mathit{Maybe}\ \mathit{Type}} \\
\mathrm{typecheck}\ (\mathsf{Var}\ \mathrm{i})\ \mathrm{ctxt} & = & \mathsf{Just}\ (\mathrm{ctxt}\ !!\ \mathrm{i}) \\
\mathrm{typecheck}\ (\mathsf{Lam}\ \mathrm{ty}\ \mathrm{tm})\ \mathrm{ctxt} & = & \textbf{case}\ \mathrm{typecheck}\ \mathrm{tm}\ (\mathrm{ty{:}ctxt})\ \textbf{of} \\
 & & \quad \mathsf{Just}\ \mathrm{ty}' \rightarrow \mathsf{Just}\ (\mathrm{ty} \Rightarrow \mathrm{ty}') \\
 & & \quad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\mathrm{typecheck}\ (\mathsf{App}\ \mathrm{tm}_1\ \mathrm{tm}_2)\ \mathrm{ctxt} & = & \textbf{case}\ \mathrm{typecheck}\ \mathrm{tm}_1\ \mathrm{ctxt}\ \textbf{of} \\
 & & \quad \mathsf{Just}\ (\mathrm{ty}_1 \Rightarrow \mathrm{ty}_2) \rightarrow \\
 & & \qquad \textbf{case}\ \mathrm{typecheck}\ \mathrm{tm}_2\ \mathrm{ctxt}\ \textbf{of} \\
 & & \qquad\quad \mathsf{Just}\ \mathrm{ty}_1' \mid \mathrm{ty}_1 \equiv \mathrm{ty}_1' \rightarrow \mathsf{Just}\ \mathrm{ty}_2 \\
 & & \qquad\quad \_ \rightarrow \mathsf{Nothing} \\
 & & \quad \_ \rightarrow \mathsf{Nothing}
\end{array}
$$

Let’s take a typechecker to bits

Let's take a Typechecker to bits

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

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

Let's take a Typechecker to bits

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

type TypeChecker = [Type] → Maybe Type

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

Let's take a Typechecker to bits

```
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 tc (ty:ctxt) of
              Just ty′ → Just (ty ⇒ ty′);    Nothing → Nothing
```

Let's take a Typechecker to bits

```
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 tc (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
```

Let's take a Typechecker to bits

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

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

Let's take a Typechecker to bits

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

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

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

Let's take a Typechecker to bits

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

type *TypeChecker* = [*Type*] → *Maybe Type*

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

typecheck :: *Term* → *TypeChecker*

Let's take a Typechecker to bits

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

type TypeChecker = [Type] → Maybe Type

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

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

Let's take a Typechecker to bits

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

type TypeChecker = [Type] → Maybe Type

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

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)

type TypeChecker = [Type] → Maybe Type

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

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)
```

Typechecker Scripts

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

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

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

More Typechecker Bits

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

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

More Typechecker Bits

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

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

$$\mathrm{failure} :: \mathit{TypeChecker}$$
$$\mathrm{failure} = \lambda \mathrm{ctxt} \to \mathsf{Nothing}$$

More Typechecker Bits

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

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

$\mathrm{failure} :: \mathit{TypeChecker}$
$\mathrm{failure} = \lambda \mathrm{ctxt} \rightarrow \mathsf{Nothing}$

$\mathrm{have} :: \mathit{Int} \rightarrow \mathit{Type} \rightarrow \mathit{TypeChecker} \rightarrow \mathit{TypeChecker}$
$\mathrm{have\ i\ ty\ tc} = \lambda \mathrm{ctxt} \rightarrow \textbf{if}\ \mathrm{ctxt}\ !!\ \mathrm{i} \equiv \mathrm{ty}\ \textbf{then}\ \mathrm{tc\ ctxt}\ \textbf{else}\ \mathsf{Nothing}$

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

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

Typechecker Scripts

A term, with an assertion

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

A term with a hole, with an assertion

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

Typechecker Scripts

A term, with an assertion

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

A term with a hole, with an assertion

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

A term with a hole, with two assertions

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

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

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

$$
\begin{array}{l}
\textbf{class}\ \mathrm{TypeChecker}\ \alpha\ \textbf{where} \\
\quad \begin{array}{lll@{\qquad}lll}
\mathrm{var} & :: & \mathit{Int} \to \alpha; & \mathrm{have} & :: & \mathit{Int} \to \mathit{Type} \to \alpha \to \alpha \\
\mathrm{lam} & :: & \mathit{Type} \to \alpha \to \alpha; & \mathrm{hasType} & :: & \mathit{Type} \to \alpha \to \alpha \\
\mathrm{app} & :: & \alpha \to \alpha \to \alpha; & \mathrm{failure} & :: & \alpha
\end{array}
\end{array}
$$

Equations 1: *Failure is contagious*

$$
\begin{array}{rcl}
\mathrm{failure} & = & \mathrm{lam}\ A\ \mathrm{failure} \\
 & = & \mathrm{app}\ \mathrm{failure}\ x \\
 & = & \mathrm{app}\ x\ \mathrm{failure} \\
 & = & \mathrm{have}\ i\ A\ \mathrm{failure} \\
 & = & \mathrm{hasType}\ A\ \mathrm{failure}
\end{array}
$$

The Algebraic Theory of Typechecking

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

class TypeChecker α where
  var :: Int → α;               have    :: Int → Type → α → α
  lam :: Type → α → α;          hasType :: Type → α → α
  app :: α → α → α;             failure :: α
```

Equations 2: *To have and to have not*

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

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

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*

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

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

Equations vs. Actual Typing

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

Theorem

$$\vdash t : A$$

if and only if

$$\text{hasType}\ A\ \llbracket t \rrbracket \neq \text{failure}$$

(in the Typechecker theory)

Have an Algebraic Theory? Think "Monad!"

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

$$
\begin{aligned}
& (\gg\!=) :: \mathit{TCTerm}\ \alpha \to (\alpha \to \mathit{TCTerm}\ \beta) \to \mathit{TCTerm}\ \beta \\
& \mathsf{Return\ a} && \gg\!= \mathsf{f} = \mathsf{f\ a} \\
& \mathsf{Var\ i} && \gg\!= \mathsf{f} = \mathsf{Var\ i} \\
& \mathsf{Lam\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Lam\ ty}\ \ (\mathsf{t} \gg\!= \mathsf{f}) \\
& \mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2 && \gg\!= \mathsf{f} = \mathsf{App}\ \ (\mathsf{t}_1 \gg\!= \mathsf{f})\ \ (\mathsf{t}_2 \gg\!= \mathsf{f}) \\
& \mathsf{Have\ i\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{Have\ i\ ty}\ \ (\mathsf{t} \gg\!= \mathsf{f}) \\
& \mathsf{HasType\ ty\ t} && \gg\!= \mathsf{f} = \mathsf{HasType\ ty}\ \ (\mathsf{t} \gg\!= \mathsf{f}) \\
& \mathsf{Failure} && \gg\!= \mathsf{f} = \mathsf{Failure}
\end{aligned}
$$

Typechecker Scripts

For, any α, $\mathit{TCTerm}\ \alpha$ is a free Typechecker algebra
(rules shall be imposed later)

Some algebraic operations, and generic effects:

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

A typechecker script, monadic style:

$$\begin{aligned} \textbf{do}\ \ &\mathrm{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B}) \\ &\mathrm{lam}\ (\mathsf{A} \Rightarrow \mathsf{B}) \\ &\mathrm{lam}\ \mathsf{A} \\ &\mathrm{have}\ 1\ (\mathsf{A} \Rightarrow \mathsf{B}) \\ &\mathrm{have}\ 0\ \mathsf{A} \\ &\mathrm{goalIs}\ \mathsf{B} \\ &\mathrm{app}\ (\mathrm{var}\ 1)\ (\mathrm{var}\ 0) \end{aligned}$$

Sorting out Scoping, Method A: de Bruijn

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

$$\textbf{class}\ \mathrm{TypeChecker}\ (\alpha :: \mathit{Nat} \to *)\ \textbf{where}$$

$$\begin{array}{llll} \mathrm{var} :: \mathit{Fin}\ n \to \alpha\ n; & \mathrm{have} & :: & \mathit{Fin}\ n \to \mathit{Type} \to \alpha\ n \to \alpha\ n \\ \mathrm{lam} :: \mathit{Type} \to \alpha\ (\mathrm{Suc}\ n) \to \alpha\ n; & \mathrm{hasType} & :: & \mathit{Type} \to \alpha\ n \to \alpha\ n \\ \mathrm{app} :: \alpha\ n \to \alpha\ n \to \alpha\ n; & \mathrm{failure} & :: & \alpha\ n \end{array}$$

(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

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

$$\begin{array}{l}\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \alpha\ \textbf{where}\\ \quad \begin{array}{llll} \mathrm{var} :: \nu \to \alpha; & \mathrm{have} & :: & \nu \to \mathit{Type} \to \alpha \to \alpha \\ \mathrm{lam} :: \mathit{Type} \to (\nu \to \alpha) \to \alpha; & \mathrm{hasType} & :: & \mathit{Type} \to \alpha \to \alpha \\ \mathrm{app} :: \alpha \to \alpha \to \alpha; & \mathrm{failure} & :: & \alpha \end{array}\end{array}$$

The abstract type ν represents variables.

Using a lam gets us a new variable.

Typecheckers with no free variables are represented by the type:

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

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

Equations???

Have an Algebraic Theory? Think "Monad!"

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\ggg=) :: \mathit{TCTerm}\ \nu\ \alpha \to (\alpha \to \mathit{TCTerm}\ \nu\ \beta) \to \mathit{TCTerm}\ \nu\ \beta} \\
\mathsf{Return\ a} & \ggg= \mathsf{f} = \mathsf{f\ a} \\
\mathsf{Var\ v} & \ggg= \mathsf{f} = \mathsf{Var\ v} \\
\mathsf{Lam\ ty\ t} & \ggg= \mathsf{f} = \mathsf{Lam\ ty}\ \ (\lambda \mathsf{v}.\ \mathsf{t\ v} \ggg= \mathsf{f}) \\
\mathsf{App}\ \mathsf{t}_1\ \mathsf{t}_2 & \ggg= \mathsf{f} = \mathsf{App}\ \ (\mathsf{t}_1 \ggg= \mathsf{f})\ \ (\mathsf{t}_2 \ggg= \mathsf{f}) \\
\mathsf{Have\ v\ ty\ t} & \ggg= \mathsf{f} = \mathsf{Have\ v\ ty}\ \ (\mathsf{t} \ggg= \mathsf{f}) \\
\mathsf{HasType\ ty\ t} & \ggg= \mathsf{f} = \mathsf{HasType\ ty}\ \ (\mathsf{t} \ggg= \mathsf{f}) \\
\mathsf{Failure} & \ggg= \mathsf{f} = \mathsf{Failure}
\end{array}
$$

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

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

A typechecker script, HOAS monadic style:

$$
\begin{array}{ll}
\textbf{do} & \mathrm{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B}) \\
& v_1 \leftarrow \mathrm{lam}\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
& v_2 \leftarrow \mathrm{lam}\ \mathsf{A} \\
& \mathrm{have}\ v_1\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
& \mathrm{have}\ v_2\ \mathsf{A} \\
& \mathrm{goalIs}\ \mathsf{B} \\
& \mathrm{app}\ (\mathrm{var}\ v_1)\ (\mathrm{var}\ v_2)
\end{array}
$$

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

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

A typechecker script, HOAS monadic style:

do goalIs $((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B})$
$\mathrm{v}_1 \leftarrow$ introduce $(\mathsf{A} \Rightarrow \mathsf{B})$
$\mathrm{v}_2 \leftarrow$ introduce A
have v_1 $(\mathsf{A} \Rightarrow \mathsf{B})$
have v_2 A
goalIs B
app (var v_1) (var v_2)

HOAS Typechecker Scripts

Some algebraic operations, and generic effects:

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

A typechecker script, HOAS monadic style:

$$\begin{array}{ll}
\mathbf{do} & \mathrm{goalIs}\ ((\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathsf{A} \Rightarrow \mathsf{B}) \\
& \mathrm{v}_1 \leftarrow \mathrm{introduce}\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
& \mathrm{v}_2 \leftarrow \mathrm{introduce}\ \mathsf{A} \\
& \mathrm{have}\ \mathrm{v}_1\ (\mathsf{A} \Rightarrow \mathsf{B}) \\
& \mathrm{have}\ \mathrm{v}_2\ \mathsf{A} \\
& \mathrm{goalIs}\ \mathsf{B} \\
& \mathrm{app}\ (\mathrm{assumption}\ \mathrm{v}_1)\ (\mathrm{assumption}\ \mathrm{v}_2)
\end{array}$$

Evaluating Typechecker Scripts

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

Evaluating Typechecker Scripts

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

Type checking:

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

Evaluating Typechecker Scripts

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

Type checking:

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

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

$$\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where}$$

$$\begin{aligned}
&\mathrm{var} && :: \nu \to \alpha\ \mathsf{SYN} \\
&\mathrm{lam} && :: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
&\mathrm{app} && :: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}
\end{aligned}$$

Bidirectional Checking and Synthesis

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

$\textbf{class}\ \mathrm{TypeChecker}\ \nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)\ \textbf{where}$

$$
\begin{array}{ll}
\mathrm{var} & :: \nu \to \alpha\ \mathsf{SYN} \\
\mathrm{lam} & :: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK} \\
\mathrm{app} & :: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN} \\
 & \\
\mathrm{switch} & :: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \\
\mathrm{ascribe} & :: \mathit{Type} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}
\end{array}
$$

Bidirectional Checking and Synthesis

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

class TypeChecker ν (α :: {CHK, SYN} → *) where
  var      :: ν → α SYN
  lam      :: (ν → α CHK) → α CHK
  app      :: α SYN → α CHK → α SYN

  switch   :: α SYN → α CHK
  ascribe  :: Type → α CHK → α SYN

  have     :: ν → (Type → α δ) → α δ
  goalIs   :: (Type → α CHK) → α CHK
  failure  :: α δ
```

Bidirectional Checking and Synthesis

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

class TypeChecker $\nu\ (\alpha :: \{\mathsf{CHK}, \mathsf{SYN}\} \to *)$ **where**

var	$:: \nu \to \alpha\ \mathsf{SYN}$
lam	$:: (\nu \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK}$
app	$:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}$
switch	$:: \alpha\ \mathsf{SYN} \to \alpha\ \mathsf{CHK}$
ascribe	$:: \textit{Type} \to \alpha\ \mathsf{CHK} \to \alpha\ \mathsf{SYN}$
have	$:: \nu \to (\textit{Type} \to \alpha\ \delta) \to \alpha\ \delta$
goalIs	$:: (\textit{Type} \to \alpha\ \mathsf{CHK}) \to \alpha\ \mathsf{CHK}$
failure	$:: \alpha\ \delta$

instance TypeChecker

$(\mathsf{CHK} \mapsto [\textit{Type}] \to \textit{Type} \to \textit{Bool}; \mathsf{SYN} \mapsto [\textit{Type}] \to \textit{Maybe Type})$

where...

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)

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

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \mu\ \alpha\ \textbf{where}} \\ \quad \mathrm{var} & :: \nu \to \alpha \\ \quad \mathrm{lam} & :: (\nu \to \alpha) \to \alpha \\ \quad \mathrm{app} & :: \alpha \to \alpha \to \alpha \end{array}$$

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 $\nu\ \mu\ \alpha$ **where**

```
var       :: ν → α
lam       :: (ν → α) → α
app       :: α → α → α
newMVar   :: (μ → α) → α
unify     :: Type μ → Type μ → α → α
```

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 $\nu\ \mu\ \alpha$ **where**

```
var       :: ν → α
lam       :: (ν → α) → α
app       :: α → α → α
newMVar   :: (μ → α) → α
unify     :: Type μ → Type μ → α → α
have      :: ([(ν, Type μ)] → α) → α
goalIs    :: (Type μ → α) → α
```

Type Inference (in Prolog)

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

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ \mathrm{TypeChecker}\ \nu\ \mu\ \alpha\ \textbf{where}} \\
\quad \mathrm{var} & :: & \nu \to \alpha \\
\quad \mathrm{lam} & :: & (\nu \to \alpha) \to \alpha \\
\quad \mathrm{app} & :: & \alpha \to \alpha \to \alpha \\
\quad \mathrm{newMVar} & :: & (\mu \to \alpha) \to \alpha \\
\quad \mathrm{unify} & :: & \mathit{Type}\ \mu \to \mathit{Type}\ \mu \to \alpha \to \alpha \\
\quad \mathrm{have} & :: & ([(\nu, \mathit{Type}\ \mu)] \to \alpha) \to \alpha \\
\quad \mathrm{goalIs} & :: & (\mathit{Type}\ \mu \to \alpha) \to \alpha \\
\quad \mathrm{failure} & :: & \alpha \\
\quad \mathrm{choice} & :: & \alpha \to \alpha \to \alpha
\end{array}
$$

Type Inference (in Prolog)

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

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

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

instance TypeChecker $(\mathit{Int} \to \mathit{Int})\ \mathit{MV}$
$(\mathit{MetaContext} \to [\mathit{Type\ MV}] \to \mathit{Type\ MV} \to \mathit{Bool})$ **where**...

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?