The Hindley-Milner type system
Hindley-Milner type system: Syntax

\[
\langle \text{term} \rangle \ ::= \begin{array}{l}
\langle \text{var} \rangle \\
\langle \text{term} \rangle \langle \text{term} \rangle \\
'\lambda' \langle \text{var} \rangle '\rightarrow' \langle \text{term} \rangle \\
'\text{let}' \langle \text{definition} \rangle \ldots \langle \text{definition} \rangle '\text{in}' \langle \text{term} \rangle
\end{array}
\]

\[
\langle \text{var} \rangle \ ::= \begin{array}{l}
'x' \\
\ldots
\end{array}
\]

\[
\langle \text{definition} \rangle \ ::= \begin{array}{l}
\langle \text{var} \rangle '==' \langle \text{term} \rangle
\end{array}
\]
Hindley-Milner type system: Syntax

\[
\begin{align*}
\langle \text{term} \rangle & ::= \langle \text{var} \rangle \\
& \quad | \langle \text{term} \rangle \langle \text{term} \rangle \\
& \quad | \lambda \langle \text{var} \rangle \mapsto \langle \text{term} \rangle \\
& \quad | \text{let} \langle \text{definition} \rangle \ldots \langle \text{definition} \rangle \text{in} \langle \text{term} \rangle \\
& \quad | \langle \text{data-con} \rangle \\
& \quad | \text{case} \langle \text{term} \rangle \text{of} \langle \text{alternative} \rangle \ldots \langle \text{alternative} \rangle \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{var} \rangle & ::= \ 'x' \ | \ldots \\
\langle \text{definition} \rangle & ::= \langle \text{var} \rangle \ '==' \langle \text{term} \rangle \\
\langle \text{data-con} \rangle & ::= \ 'K' \ | \ldots \\
\langle \text{alternative} \rangle & ::= \langle \text{pat} \rangle \ '\mapsto' \langle \text{term} \rangle \\
\langle \text{pat} \rangle & ::= \langle \text{data-con} \rangle \langle \text{pat} \rangle \ldots \langle \text{pat} \rangle \\
& \quad | \langle \text{var} \rangle \ | \ '_'
\end{align*}
\]
Hindley-Milner type system: Types

\[\langle \sigma\text{-type} \rangle ::= \forall \langle ty\text{-var} \rangle \ldots \langle ty\text{-var} \rangle \ '. ' \langle \tau\text{-type} \rangle\]

\[\langle \tau\text{-type} \rangle ::= \langle ty\text{-var} \rangle\]
\[\quad \mid \langle \tau\text{-type} \rangle \rightarrow \langle \tau\text{-type} \rangle\]

\[\langle ty\text{-var} \rangle ::= \alpha \mid \ldots\]
Hindley-Milner type system: Types

\[\langle \sigma\text{-type} \rangle ::= \ '\forall' \ \langle \text{ty-var} \rangle \ ... \ \langle \text{ty-var} \rangle ' . ' \ \langle \tau\text{-type} \rangle\]

\[\langle \tau\text{-type} \rangle ::= \ \langle \text{ty-var} \rangle\]
\[\ | \ \langle \tau\text{-type} \rangle ' \rightarrow' \ \langle \tau\text{-type} \rangle\]
\[\ | \ \langle \text{ty-con} \rangle \ \langle \tau\text{-type} \rangle \ ... \ \langle \tau\text{-type} \rangle\]

\[\langle \text{ty-var} \rangle ::= \ '\alpha' \ | \ ...\]

\[\langle \text{ty-con} \rangle ::= \ 'T' \ | \ ...\]
Hindley-Milner type system: Derivation rules

\[
\begin{align*}
x &::= \sigma \in \Gamma \quad \tau \in \text{Inst}(\sigma) \\
\Gamma \vdash x :: \tau & \quad \text{(VAR)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash F :: \tau_1 \to \tau_2 \quad \Gamma \vdash E :: \tau_1 \\
\Gamma \vdash FE :: \tau_2 & \quad \text{(APP)}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x :: \tau_1 &\vdash E :: \tau_2 \\
\Gamma \vdash \lambda x \mapsto E :: \tau_1 \to \tau_2 & \quad \text{(LAM)}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x :: \tau_0 &\vdash E_0 :: \tau_0 \\
\sigma &\equiv \text{Gen}(\Gamma, \tau_0) \\
\Gamma, x :: \sigma &\vdash E :: \tau \\
\Gamma \vdash \text{let } x = E_0 \text{ in } E :: \tau & \quad \text{(LET)}
\end{align*}
\]
Hindley-Milner type system: Derivation rules

\[
\frac{x :: \sigma \in \Gamma \quad \tau \in \text{Inst}(\sigma)}{\Gamma \vdash x :: \tau} \quad \text{(VAR)}
\]

\[
\frac{\Gamma \vdash F :: \tau_1 \to \tau_2 \quad \Gamma \vdash E :: \tau_1}{\Gamma \vdash FE :: \tau_2} \quad \text{(APP)}
\]

\[
\frac{\Gamma, x :: \tau_1 \vdash E :: \tau_2}{\Gamma \vdash \lambda x \mapsto E :: \tau_1 \to \tau_2} \quad \text{(LAM)}
\]

\[
\frac{\Gamma, x :: \tau_0 \vdash E_0 :: \tau_0 \quad \sigma = \text{Gen}(\Gamma, \tau_0) \quad \Gamma, x :: \sigma \vdash E :: \tau}{\Gamma \vdash \text{let } x = E_0 \text{ in } E :: \tau} \quad \text{(LET)}
\]

\[
\tau \text{ in VAR?}
\]

\[
\tau_1 \text{ in LAM?}
\]

\[
\tau_1 \text{ in LET?}
\]
\[ \mathcal{W}(\Gamma, E) = (\Sigma, \tau) \]

where

- \( \Gamma \) : a type context, mapping variables to types
- \( E \) : the expression whose type we are to infer
- \( \Sigma \) : a substitution, mapping type variables to types
- \( \tau \) : the inferred type of \( E \)
HM type inference algorithms

\[ \mathcal{W} \]

\[ \mathcal{W}(\Gamma, E) = (\Sigma, \tau) \]

where

\[ \Gamma : a \text{ type context, mapping variables to types} \]

\[ E : the \text{ expression whose type we are to infer} \]

\[ \Sigma : a \text{ substitution, mapping type variables to types} \]

\[ \tau : the \text{ inferred type of } E \]

\[ \mathcal{M} \]

\[ \mathcal{M}(\Gamma, E, \tau) = \Sigma \]

where

\[ \Gamma : a \text{ type context, mapping variables to types} \]

\[ E : the \text{ expression to typecheck} \]

\[ \tau : the \text{ expected type of } E \]

\[ \Sigma : a \text{ substitution, mapping type variables to types} \]
Hindley-Milner is linear
\[ \mathcal{W} \text{ for application} \]

\[ \mathcal{W}(\Gamma, E F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma \beta) \]

where

\[ (\Sigma_1, \tau_1) = \mathcal{W}(\Gamma, E) \]

\[ (\Sigma_2, \tau_2) = \mathcal{W}(\Sigma_1 \Gamma, F) \]

\[ \Sigma = \mathcal{U}(\Sigma_2 \tau_1 \sim \tau_2 \rightarrow \beta) \]

\[ \beta \text{ fresh} \]
Linearity

\[ \mathcal{W}(\Gamma, E F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma \beta) \]

where

\[
\begin{align*}
(\Sigma_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Sigma_2, \tau_2) &= \mathcal{W}(\Sigma_1 \Gamma, F) \\
\Sigma &= \mathcal{U}(\Sigma_2 \tau_1 \sim \tau_2 \rightarrow \beta) \\
\beta & \text{ fresh}
\end{align*}
\]
\[ \mathcal{V}(\Gamma, E F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma \beta) \]

where

\[ (\Sigma_1, \tau_1) = \mathcal{V}(\Gamma, E) \]
\[ (\Sigma_2, \tau_2) = \mathcal{V}(\Sigma_1 \Gamma, F) \]
\[ \Sigma = \mathcal{U}(\Sigma_2 \tau_1 \sim \tau_2 \rightarrow \beta) \]

\( \beta \) fresh
Linearity

\( \mathcal{W} \) for application

\( \mathcal{W}(\Gamma, E F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma \beta) \)

where

\( (\Sigma_1, \tau_1) = \mathcal{W}(\Gamma, E) \)

\( (\Sigma_2, \tau_2) = \mathcal{W}(\Sigma_1 \Gamma, F) \)

\( \Sigma = \mathcal{U}(\Sigma_2 \tau_1 \sim \tau_2 \to \beta) \)

\( \beta \) fresh
$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma \beta)$

where

$(\Sigma_1, \tau_1) = \mathcal{W}(\Gamma, E)$

$(\Sigma_2, \tau_2) = \mathcal{W}(\Sigma_1 \Gamma, F)$

$\Sigma = \mathcal{U}(\Sigma_2 \tau_1 \sim \tau_2 \rightarrow \beta)$

$\beta$ fresh
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
Error messages

Input

isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)

Output from GHC (7.10.3)

foo.hs:1:24:
    Couldn't match expected type `Bool'
    with actual type `Maybe a'
    In the first argument of `not', namely `x'
    In the expression: not x
Error messages

Input

isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)

Output from GHC (7.10.3)

foo.hs:1:24:
   Couldn't match expected type `Bool'
       with actual type `Maybe a'
   In the first argument of `not', namely `x'
   In the expression: not x
**Error messages**

**Input**

```haskell
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

**Output from Hugs 98 (September 2006)**

```
ERROR "foo.hs":1 - Type error in application
*** Expression : isJust x
*** Term : x
*** Type : Bool
*** Does not match : Maybe a
```
Input

isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)

Output from Hugs 98 (September 2006)

ERROR "foo.hs":1 - Type error in application
*** Expression : isJust x
*** Term : x
*** Type : Bool
*** Does not match : Maybe a
isJust :: Maybe a -> Bool
not :: Bool -> Bool

foo x = (isJust x, not x)

So where is the error?
A compositional type system for HM
Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions.
- The context of a variable occurrence can affect the type of some enclosing scope.

```haskell
foo x = (isJust x, not x)
```
To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions.

The context of a variable occurrence can affect the type of some enclosing scope.

```haskell
foo x = (isJust x, not x)

isJust x :: Bool
x :: Maybe α
```
To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions.

The context of a variable occurrence can affect the type of some enclosing scope.

foo x = (isJust x, not x)

\[
\begin{align*}
\text{not } x &:: \text{Bool} \\
x &:: \text{Bool}
\end{align*}
\]
To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions.

The context of a variable occurrence can affect the type of some enclosing scope:

```haskell
foo x = (isJust x, not x)

isJust x :: Bool  not x :: Bool
x :: Maybe α  ⇒⇐  x :: Bool
```
To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions.

The context of a variable occurrence can affect the type of some enclosing scope:

```haskell
foo x = (isJust x, not x)

isJust x :: Bool    not x :: Bool
x :: Maybe α    =><⇒    x :: Bool
```

So we will assign to subexpressions, instead of types, something called *typings*:

```haskell
isJust x :: \{ x :: Maybe α \} ↰ Bool
not x :: \{ x :: Bool \} ↰ Bool
```
Compositional derivation rules

\[
\frac{(x :: \Delta_0 \vdash \tau_0) \in \Gamma}{x :: \Delta \vdash \tau} \quad \text{(VAR)}
\]

\[
\frac{\Gamma, (x :: \{x :: \alpha\} \vdash \alpha) \vdash E :: \Delta \vdash \tau_2 \quad \alpha \text{ fresh}}{(x :: \tau_1) \in \Delta \lor (x \notin \Delta \land \tau_1 = \alpha)}
\]

\[
\frac{}{\Gamma \vdash \lambda x \mapsto E :: \Delta \setminus x \vdash \tau_1 \rightarrow \tau_2} \quad \text{(LAM)}
\]

\[
\frac{\Gamma \vdash F :: \Delta_1 \vdash \tau_1 \quad \Gamma \vdash E :: \Delta_2 \vdash \tau_2}{\Gamma \vdash FE :: \Delta \vdash \tau} \quad \text{(APP)}
\]

where \( \alpha \) fresh

\[
(\Delta, \Sigma) = U(\Delta_1, \Delta_2, \tau_1 \sim \tau_2 \rightarrow \alpha)
\]

\[
\tau = \Sigma \alpha
\]
Compositional derivation rules: \texttt{let}

\begin{align*}
\Gamma, (x :: \{x :: \alpha\} \vdash \alpha) & \vdash E_0 :: \Delta_0 \vdash \tau_0 & \alpha \text{ fresh} \\
\Gamma, (x :: \Delta_0'' \vdash \Sigma_0 \tau_0) & \vdash E :: \Delta \vdash \tau \\
\Gamma \vdash \texttt{let} \ x = E_0 \ \texttt{in} \ E :: \Delta' \vdash \Sigma \tau
\end{align*}

(\textsc{Let})

where

\begin{align*}
(\Delta_0', \Sigma_0) &= \mathcal{U}(\Delta_0, \tau_0 \sim \Delta_0(x)) \\
\Delta_0'' &= \Delta_0 \setminus x \\
(\Delta', \Sigma) &= \mathcal{U}(\Delta_0'', \Delta)
\end{align*}
Where is \texttt{let}-polymorphism?

- If \((x :: \Delta_0 \vdash \tau_0) \in \Gamma\), then \(x\) is polymorphic iff \(x \notin \Delta_0\):

\[
\frac{(x :: \Delta_0 \vdash \tau_0) \in \Gamma \quad \Delta \vdash \tau \in \text{Freshen}(\Delta_0 \vdash \tau_0)}{\Gamma \vdash x :: \Delta \vdash \tau}
\]

This results in two occurrences of \(x\) to yield a constraint that their types match only if \(x \in \Delta \ (\leftrightarrow x \in \Delta_0)\)

- \(\lambda x \rightarrow E\) introduces \(x :: \{x :: \alpha\} \vdash \alpha\) to \(\Gamma\), i.e. \(x\) is \textit{monomorphic}

- \texttt{let} \(x = E_0\ \texttt{in} E\) introduces \(x :: \Delta \vdash \tau\) to \(\Gamma\) after removing \(x\) from the typing of \(E_0\), i.e. \(x\) is \textit{polymorphic in} \(E\)
Implementation
Both linear and compositional type checking implemented for our model language:

- Concrete syntax (parser & pretty printer)
  - Indentation-based parsing is a nightmare
  - haskell-src-exts to the rescue!
- unification-fd-based representation
  - Immediate rewriting of type-meta-variables: no delayed occurs checks
- Explicit zonking

```haskell
class (Unifiable t, Variable v, Monad m) ⇒ MonadTC t v m |

  where
  freshVar :: m v
  readVar :: v → m (Maybe (UTerm t v))
  writeVar :: v → UTerm t v → m ()
  zonk :: (Traversable t, MonadTC t v m)
    ⇒ UTerm t v → m (UTerm t v)
```
Implementation: \texttt{hm-compo}

Both linear and compositional type checking implemented for our model language:

- Code mostly shared between the two typecheckers

```haskell
data TC ctx err s loc a
instance MonadReader ctx (TC ctx err s loc)
instance MonadError err (TC ctx err s loc)
instance MonadTC Ty0 (MVar s) (TC ctx err s loc)
freshTVar :: TC ctx err s loc TVar
```

- Representation of $\Gamma$ is different: there are no $\sigma$-types in the compositional type system.
Demo time
Motivating example

Input

isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = MkPair (isJust x) (not x)

Output of hm-compo

demo/pair.hm (13,8):
    MkPair (not x) (isJust x)

Cannot unify 'Bool' with 'Maybe a' when unifying 'x':
Cannot unify 'Bool' with 'Maybe a' in the following context:
    MkPair (not x) isJust x
    Bool -> Pair Bool Bool Bool Bool
    x :: Bool Maybe a
Types agree for well-typed terms

id :: a → a
const :: a → b → a
fix :: (a → a) → a
flip :: (a → b → c) → b → a → c
foldr :: (a → b → b) → b → List a → b
map :: (a → b) → List a → List b
undefined :: a
undefined1 :: a
undefined2 :: a
For further information

- *Compositional Explanation of Types and Algorithmic Debugging of Type Errors*, Olaf Chitil (2001)
- *Compositional Type Checking for Hindley-Milner Type Systems with Ad-hoc Polymorphism*, Gergő Érdi (2011)