We discussed type checking rules for an explicitly typed language.

It is annoying to be forced to explicitly type every binding.
We discussed type checking rules for an explicitly typed language.

It is annoying to be forced to explicitly type every binding.

Can the compiler infer the type of each expression?
We discussed type checking rules for an explicitly typed language.

It is annoying to be forced to explicitly type every binding.

Can the compiler infer the type of each expression?

Can we find a set of type inference rules describing an inference algorithm?
We discussed **type checking** rules for an explicitly typed language.

It is annoying to be forced to explicitly type every binding.

Can the compiler **infer** the type of each expression?

Can we find a set of type inference rules describing an inference algorithm?

We need to answer two questions first:

- Under which circumstances does a set of inference rules describe an algorithm?
- Which type should be inferred for a polymorphic expression?
Consider the rules for the static semantics (typing) or big-step semantics of MinHs:

- the expression and the (type) environment can be interpreted as the *input*, the value/type as the *output*
- the rules are *syntax directed*: only a single rule applies for each syntactic construct
What is the type of the function:

```ocaml
letfun f x = (fst x) + 1
```

Some possible types:

1. `(Int; Int) -> Int`
2. `(Int; Bool) -> Int`
3. `(Int; (Int; Int)) -> Int`
4. `8 a : (Int; a) -> Int`

The first three types are special cases of the fourth type.

Accordingly, we write $0$ if $0$ is less general than $\varepsilon$, or in other words: if $\varepsilon > 0$, then for all expressions $e$, if $e > 0$, then $e = 0$.

Examples:

- `(Int; Int) -> Int`
- `8 a : (a; a) -> a`
- `8 a : (a; a) -> a`
- `8 b : (a; b) -> a`
What is the type of the function:

\[
\text{letfun } f \; x = (\text{fst } x) + 1
\]

Some possible types:

1. \((\text{Int, Int}) \rightarrow \text{Int}\)
2. \((\text{Int, Bool}) \rightarrow \text{Int}\)
3. \((\text{Int, (Int, Int)}) \rightarrow \text{Int}\)
4. \(\forall a. (\text{Int, } a) \rightarrow \text{Int}\)
PRINCIPAL TYPES

What is the type of the function:

\[ \text{letfun } f \ x = (\text{fst } x) + 1 \]

Some possible types:

1. \((\text{Int, Int}) \rightarrow \text{Int}\)
2. \((\text{Int, Bool}) \rightarrow \text{Int}\)
3. \((\text{Int, (Int, Int)}) \rightarrow \text{Int}\)
4. \(\forall a. (\text{Int}, a) \rightarrow \text{Int}\)

The first three types are special cases of the fourth type.

- we write \(\tau' \leq \tau\) if \(\tau'\) is less general than \(\tau\)
- in other words: if \(\tau' \leq \tau\), then for all expressions \(e\), if \(e : \tau\), then also \(e : \tau'\)

- examples:
  - \((\text{Int, Int}) \rightarrow \text{Int} \leq \forall a. (a, a) \rightarrow a\)
  - \(\forall a. (a, a) \rightarrow a \leq \forall a. \forall b. (a, b) \rightarrow a\)
We are interested in the least general type $\tau$ of an expression $e$ such that any $e : \tau'$ implies $\tau' \leq \tau$

This is called the principal type of the expression $e$.

The principal type of

\[
\text{letfun } f \ x = (\text{fst } x) + 1
\]

is $\forall a. (\text{Int}, a) \rightarrow \text{Int}$
Implicitly Typed MinHs

Similar to MinHs, but

- no type annotations for functions and constructors (e.g. in1)
- no recursive types: rec, roll, and unroll not part of language
- no explicit instantiation of types: Fun and inst not part of the language
- Types of built-in functions are in the type enviroment:

\[ \Gamma = \{ + : (\text{Int, Int}) \to \text{Int}, \ldots, \text{fst} : \forall a. \forall b. (a, b) \to a, \ldots \} \]
What is the type of the following expressions:

$\Rightarrow \text{inl (True)}$

$\Rightarrow \text{fst (1, True)}$

$\Rightarrow \text{roll (inl (1))} \text{ (if recursive types were part of the language)}$
What is the type of the following expressions:

- \texttt{inl (True)}
  - we would not be able to type it in a monomorphic setting

- \texttt{fst (1, True)}

- \texttt{roll (inl (1))} (if recursive types were part of the language)
What is the type of the following expressions:

- \texttt{inl} (True)
  - we would not be able to type it in a monomorphic setting
  - polymorphic type: \( \forall a. (\text{Bool}, a) \)

- \texttt{fst} (1, True)

- \texttt{roll} (\texttt{inl} (1)) (if recursive types were part of the language)
What is the type of the following expressions:

\[ \text{inl (True)} \]
- we would not be able to type it in a monomorphic setting
- polymorphic type: \( \forall a. (Bool, a) \)

\[ \text{fst (1, True)} \]
- type of \( \text{fst} : \forall a. \forall b. (a, b) \rightarrow a \)
- type of \( (1, \text{True}) : (\text{Int}, \text{Bool}) \)
- type of \( \text{fst (1, True)} \):

\[ \text{roll (inl (1))} \] (if recursive types were part of the language)
What is the type of the following expressions:

$\text{⇒ } \text{inl (True)}$

- we would not be able to type it in a monomorphic setting
- polymorphic type: $\forall a.(\text{Bool}, a)$

$\text{⇒ } \text{fst (1, True)}$

- type of $\text{fst}$: $\forall a. \forall b. (a, b) \rightarrow a$
- type of $(1, \text{True})$: $(\text{Int}, \text{Bool})$
- type of $\text{fst (1, True)}$: Int

$\text{⇒ } \text{roll (inl (1))}$ (if recursive types were part of the language)

- we cannot type it, therefore not in part of the language we are considering
Typing Rules

- Variables and application stays the same:

\[
\begin{align*}
\Gamma \vdash x : \tau & \quad \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash x : \tau & \quad \Gamma \vdash \text{apply}(e_1, e_2) : \tau_2
\end{align*}
\]

\[
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2
\]

\[
\Gamma \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)
\]

- \text{inr} and \text{inl} typing rules introduce free type variables:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash \text{inl}(e_1) : (\tau_1 + \tau_2) \\
\Gamma \vdash e_2 : \tau_2 & \quad \Gamma \vdash \text{inr}(e_2) : (\tau_1 + \tau_2)
\end{align*}
\]

- Functions:

\[
\Gamma \vdash \text{fun}(f.x.e) : \tau_1 \rightarrow \tau_2
\]
\( \forall \)-introduction and elimination:

\[
\begin{align*}
\Gamma &\vdash e : \forall t.\tau \\
\Gamma &\vdash e : [\tau_1/t]_{\tau}
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash e : \tau \quad t \notin TV(\Gamma) \\
\Gamma &\vdash e : \forall t.\tau
\end{align*}
\]

Where \( TV(\Gamma) \) is the set of all type variables occurring in \( \Gamma \).
Are the inference rules syntax directed?

Can we view $I$ and the expression as input, the type as output?
Are the inference rules syntax directed?

- no — ∀-introduction rule can always be applied

Can we view \( \Gamma \) and the expression as input, the type as output?
Are the inference rules syntax directed?

- no — $\forall$-introduction rule can always be applied

Can we view $I$ and the expression as input, the type as output?

- no — $\forall$-elimination rule may instantiate an expression to the “wrong” type
Are the inference rules syntax directed?

- no — ∀-introduction rule can always be applied

Can we view \( \Gamma \) and the expression as input, the type as output?

- no — ∀-elimination rule may instantiate an expression to the "wrong" type

\[
\begin{align*}
\Gamma \vdash \text{fst} : \forall a. \forall b. (a, b) \rightarrow a & : \vdash \\
\Gamma \vdash \text{fst} : (\text{Bool}, \text{Bool}) \rightarrow \text{Bool} & : \Gamma \vdash \text{pair}(1, \text{True}) : (\text{Int}, \text{Bool}) \\
\Gamma \vdash \text{apply} (\text{fst}, \text{pair}(1, \text{True})) : ?
\end{align*}
\]
Idea:

- delay instantiation until necessary
- “merge” required and computed argument type
- replace $\forall$-quantified variables by free, fresh variables
Idea:

- delay instantiation until necessary
- “merge” required and computed argument type
- replace $\forall$-quantified variables by free, fresh variables

\[
\begin{align*}
\Gamma \vdash \text{fst} : \forall a. \forall b. (a, b) \rightarrow a & : \\
\Gamma \vdash \text{fst} : (x, y) \rightarrow x & \\
\Gamma \vdash \text{pair}(1, \text{True}) : (\text{Int}, \text{Bool}) & \\
\Gamma \vdash \text{apply} (\text{fst}, \text{pair}(1, \text{True})) : ?
\end{align*}
\]
Idea:

→ delay instantiation until necessary
→ “merge” required and computed argument type
→ replace ∀-quantified variables by free, fresh variables

\[\Gamma \vdash \text{fst} : \forall a. \forall b. (a, b) \to a\]

\[\Gamma \vdash (x, y) \to x\]

\[\Gamma \vdash \text{pair}(1, \text{True}) : (\text{Int}, \text{Bool})\]

\[\Gamma \vdash \text{apply} (\text{fst}, \text{pair}(1, \text{True})) : ?\]

We can substitute any type for the free variables, therefore

\[\text{[Int}/x[\text{Bool}/y](x, y) = (\text{Int}, \text{Bool})\]
In some cases, it is necessary to substitute variables on both sides:

\[\text{Int}/a](a, \text{Bool}) = \text{Bool}/b](\text{Int}, b)\]

or to replace variables

\[(a, a) = [a/b](a, b)\]
A substitution \( S \), with \( S \tau = S \tau' \) is called a unifier of \( \tau \) and \( \tau' \). For the type inference algorithm, we need the most general unifier (mgu).

We write \( \tau \overset{S}{\sim} \tau' \) if \( S \) is a mgu of \( \tau \) and \( \tau' \).

Examples: what are the mgu’s for the following pairs of types:

- \( (a, (a, a)) \) and \( (b, c) \)
- \( \text{Int} \) and \( \text{Bool} \)
- \( (a, (a, a)) \) and \( ((a, a), a) \)
Back to the type inference algorithm:

\[
\begin{align*}
\frac{x : \forall a_1 \ldots \forall a_n. \tau \in \Gamma}{\Gamma \vdash x : [\beta_1/a_1] \ldots [\beta_n/a_n] \tau, \quad \beta_i \text{ fresh}}
\end{align*}
\]

\[
\begin{align*}
T \Gamma \vdash e_1 : \tau_1 & \quad T' \Gamma \vdash e_2 : \tau_2 & \quad T' \tau_1 \overset{U}{\sim} (\tau_2 \rightarrow \alpha) & \quad \alpha \text{ fresh} \\
\hline
UT' \Gamma \vdash \text{apply}(e_1, e_2) : U \alpha
\end{align*}
\]

\[
\begin{align*}
T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau & \quad \alpha \text{ fresh} \\
\hline
TT \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau
\end{align*}
\]

\[\Rightarrow \text{ rules are syntax directed}\]

\[\Rightarrow \Gamma, e \text{ input, unifier and type output}\]
\[
\frac{T\Gamma \vdash e_1 : \tau_1 \quad T'T\Gamma \vdash e_2 : \tau_2 \quad T'\tau_1 \overset{U}{\sim} \tau_2 \to \alpha}{UT'T\Gamma \vdash \text{apply}(e_1, e_2) : U\alpha} \quad \alpha \text{ fresh}
\]

\[
\begin{array}{l}
\text{fst} : \forall a. \forall b. (a, b) \to a \in \Gamma \quad (x, y) \to x \overset{U}{\sim} (\text{Int, Bool}) \to a \\
[[\!]\Gamma \vdash \text{fst} : (x, y) \to x \\
[[\!]\Gamma \vdash (1, \text{True}) : (\text{Int, Bool}) \\
U[[\!]\Gamma \vdash \text{apply} (\text{fst}, (1, \text{True})) : \text{Int}(= U a) \\
\text{where } U = [\text{Int}/a][\text{Bool}/y][\text{Int}/x]
\end{array}
\]
To compute the type we

① infer the type of \( \text{fst} \)

- \( \forall a. \forall b. (a, b) \rightarrow a \in T \), replace quantified variables by fresh variables \( x, y \)
- no instantiation of free variables necessary, therefore \( T \) of application rule is the empty substitution

② infer the type of \( (1, \text{True}) \)

- \( T' \) is the empty substitution

③ compute most general unifier (mgu) \( U \) of \( (x, y) \rightarrow x \) and \( (\text{Int}, \text{Bool}) \rightarrow a \) for a new free variable \( a \).
A simple function:

\[
\begin{align*}
&T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau \\
&TT \vdash \text{fun}(f.x.e) : T\alpha \to \tau \\
&\alpha \text{ fresh} \\
&\Gamma \vdash \text{fun}(f.x.(x, x)):
\end{align*}
\]
A simple function:

\[
\begin{align*}
\frac{T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau}{TT \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau} \quad \text{\(\alpha\) fresh}
\end{align*}
\]

\[
\begin{align*}
\frac{(\Gamma \cup \{x : a\}) \vdash (x, x) : \alpha}{\Gamma \vdash \text{fun}(f.x.(x, x)) :}
\end{align*}
\]
A simple function:

\[
\frac{T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau}{TT \vdash \mathit{fun}(f.x.e) : T\alpha \rightarrow \tau} \quad \alpha \text{ fresh}
\]

\[
\frac{[\ ](\Gamma \cup \{x : a\}) \vdash (x, x) : (a, a)}{\Gamma \vdash \mathit{fun}(f.x.(x, x)) :}
\]
A simple function:

\[
\frac{T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau}{TT \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau} \quad \alpha \ fresh
\]

\[
\frac{[\![\Gamma \cup \{x : a\}]\!] \vdash (x, x) : (a, a)}{[\!]\Gamma \vdash \text{fun}(f.x.(x, x)) : [\!]a \rightarrow (a, a)}
\]
\[ T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau \]
\[ TT \vdash \text{fun}(f.x.e) : T\alpha \to \tau \quad \alpha \text{ fresh} \]

\[ \Gamma \vdash \text{fun}(f.x.(x + 1, x + 1)) : \]
\[ T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau \]

\[ TT \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau \quad \alpha \text{ fresh} \]

\[ (\Gamma \cup \{x : a\}) \vdash (x + 1, x + 1) : \]

\[ \Gamma \vdash \text{fun}(f.x.(x + 1, x + 1)) : \]
\[
T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau \\
\overline{\Gamma \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau} \quad \alpha \text{ fresh}
\]

\[
[\text{Int}/a](\Gamma \cup \{x : a\}) \vdash (x + 1, x + 1) : (\text{Int}, \text{Int}) \\
\overline{\Gamma \vdash \text{fun}(f.x.(x + 1, x + 1)) :}
\]
\[ T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau \]
\[ \Gamma \vdash \text{fun}(f.x.e) : T\alpha \rightarrow \tau \quad \alpha \text{ fresh} \]

\[
[\text{Int}/a](\Gamma \cup \{x : a\}) \vdash (x + 1, x + 1) : (\text{Int}, \text{Int})
\]
\[
[\text{Int}/a] \Gamma \vdash \text{fun}(f.x.(x + 1, x + 1)) : [\text{Int}/a]a \rightarrow (\text{Int}, \text{Int})
\]