Polymorphic Types
and Type Inference
Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo
Idea

**What should be a type annotation for “universal” functions?**

⇒ ... we add variables for types.

- Syntax of type annotations:

  \[ \tau ::= \iota \mid \tau \rightarrow \tau \mid \alpha, \beta, \ldots \]

  ⇒ \alpha, \beta, \ldots are type variables

- How are the type variables used? substitutions \([\tau'/\alpha]\)
Type Variables and Substitutions

Idea

*What should be a type annotation for “universal” functions?*

⇒ *... we add variables for types.*

- Syntax of type annotations:

  \[ τ ::= ι \mid τ → τ \mid α, β, ... \]

  ⇒ \( α, β, \ldots \) are type variables

- How are the type variables used? substitutions \([τ' / α]τ\)
Type Variables and Substitutions

Idea

What should be a type annotation for “universal” functions?

⇒ ... we add variables for types.

• Syntax of type annotations:

$$\tau ::= \iota \mid \tau \to \tau \mid \alpha, \beta, \ldots$$

⇒ \(\alpha, \beta, \ldots\) are type variables

• How are the type variables used? substitutions \([\tau'/\alpha]\).
Idea

What should be a type annotation for “universal” functions?

⇒ ... we add variables for types.

- Syntax of type annotations:

\[ \tau ::= \iota \mid \tau \rightarrow \tau \mid \alpha, \beta, \ldots \]

⇒ \(\alpha, \beta, \ldots\) are type variables

- How are the type variables used?
Type Variables and Substitutions

Idea

What should be a type annotation for “universal” functions?

⇒ ... we add variables for types.

• Syntax of type annotations:

\[ \tau ::= \iota \mid \tau \to \tau \mid \alpha, \beta, \ldots \]

⇒ \(\alpha, \beta, \ldots\) are type variables

• How are the type variables used? substitutions \(\tau'/\alpha\tau\)!
What does it Mean?

- Consider an expression

\[ \lambda x : \alpha . \lambda y : \beta . (x \ (x \ y)) \]

⇒ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for \( \alpha \) and \( \beta \) (universal reading).
  2. well typed for some substitution for \( \alpha, \beta \) (existential reading).

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

- Consider an expression

\[ \lambda x : \alpha . \lambda y : \beta . (x (x y)) \]

⇒ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for \( \alpha \) and \( \beta \) (universal reading).

...and this one is not!

  2. well typed for some substitution for \( \alpha, \beta \) (existential reading).

...and this one is for \( [\beta \rightarrow \beta/\alpha] \)

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

• Consider an expression

\[ \lambda x : \alpha . \lambda y : \beta . (x (x y)) \]

⇒ is the expression well-formed (type-able)?

• two points of view:

1 well typed for all substitutions for \( \alpha \) and \( \beta \) (universal reading).

... and this one is not!

2 well typed for some substitution for \( \alpha, \beta \) (existential reading).

... and this one is for \( [\beta \rightarrow \beta/\alpha] \)

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

- Consider an expression

\[ \lambda x : \alpha . \lambda y : \beta . (x \ (x \ y)) \]

⇒ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for \( \alpha \) and \( \beta \) (*universal reading*).
     ...and this one is not!
  2. well typed for some substitution for \( \alpha, \beta \) (*existential reading*).
     ...and this one is for \([\beta \rightarrow \beta/\alpha]\) or \([\text{int} \rightarrow \text{int}/\alpha, \text{int}/\beta]\)

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

- Consider an expression

\[
\lambda x : \alpha . \lambda y : \beta . (x \ (x \ y))
\]

⇒ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for \(\alpha\) and \(\beta\) (*universal reading*).
      ...and this one is not!
  2. well typed for some substitution for \(\alpha, \beta\) (*existential reading*).
      ...and this one is for \([\beta \to \beta/\alpha]\) or \([\text{int} \to \text{int}/\alpha, \text{int}/\beta]\)

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

• Consider an expression

\[ \lambda x : \alpha \cdot \lambda y : \beta \cdot (x \ (x \ y)) \]

⇒ is the expression well-formed (type-able)?

• two points of view:

1. well typed for all substitutions for \( \alpha \) and \( \beta \) (*universal reading*).

   ... and this one is not!

2. well typed for some substitution for \( \alpha, \beta \) (*existential reading*).

   ... and this one is for \([\beta \rightarrow \beta/\alpha]\) or \([\text{int} \rightarrow \text{int}/ \alpha, \text{int}/ \beta]\)

Idea

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
What does it Mean?

- Consider an expression

\[ \lambda x : \alpha . \lambda y : \beta . (x \ (x \ y)) \]

⇒ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for \( \alpha \) and \( \beta \) (*universal reading*).
     ... and this one is not!
  2. well typed for some substitution for \( \alpha, \beta \) (*existential reading*).
     ... and this one is for \([\beta \rightarrow \beta/\alpha]\) or \([\text{int} \rightarrow \text{int}/\alpha, \text{int}/\beta]\)

**Idea**

We use type variables and substitutions “as general as possible” to infer/reconstruct type annotations.
Let Polymorphism

This is not quite good enough:

- consider the following expression:

  \[
  \text{let } d = \lambda f : \alpha \to \alpha . \lambda x : \alpha . (f (f x)) \text{ in }
  \]

  \[
  \text{let } a = d (\lambda x : \text{int} . x + 1) \ 2 \text{ in }
  \]

  \[
  \text{let } a = d (\lambda x : \text{str} . x ^ x) \ "foo" \text{ in} \ldots
  \]

- What is the type of \(d\)? (i.e., what do we substitute for \(\alpha\)?)

\[
\[ \introduces \ \text{we need a different type tag for } d \]
\]

Idea

We need to be able to substitute different types for type variables.

\[
d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha
\]

\[
\introduces \ \text{introduced by the let construct (hence the name)}
\]
Let-Polymorphism

This is not quite good enough:

- consider the following expression:

\[
\text{let } d = \lambda f : \alpha \to \alpha. \lambda x : \alpha. (f (f x)) \text{ in }
\]

\[
\text{let } a = d (\lambda x : \text{int}. x + 1) 2 \text{ in }
\]

\[
\text{let } a = d (\lambda x : \text{str}. x \hat{x}) "foo" \text{ in } \ldots
\]

- What is the type of \(d'\)? (i.e., what do we substitute for \(\alpha\)?)

\[
\ldots \text{we need a different type tag for } d!
\]

**Idea**

We need to be able to substitute different types for type variables.

\[
d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha
\]

introduced by the let construct (hence the name)
Let-Polymorphism

This is not quite good enough:

• consider the following expression:

\[
\text{let } d = \lambda f : \alpha \rightarrow \alpha . \lambda x : \alpha . (f (f x)) \text{ in }
\]

\[
\text{let } a = d (\lambda x : \text{int}. x + 1) \text{ in }
\]

\[
\text{let } a = d (\lambda x : \text{str}. x \hat{x}) \text{ "foo" in...}
\]

• What is the type of \(d'\)? (i.e., what do we substitute for \(\alpha\)?)

\[
\ldots \text{we need a different type tag for } d!
\]

Idea

We need to be able to substitute different types for type variables.

\[
d : \forall \alpha : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha
\]

\(\Rightarrow\) introduced by the \text{let} construct (hence the name)
Let-Polymorphism

This is not quite good enough:

- consider the following expression:

\[
\text{let } d = \lambda f : \alpha \to \alpha . \lambda x : \alpha . (f (f x)) \text{ in }
\]
\[
\text{let } a = d (\lambda x : \text{int}.x + 1) 2 \text{ in }
\]
\[
\text{let } a = d (\lambda x : \text{str}.x \hat{x}) "foo" \text{ in } \ldots
\]

- What is the type of \(d'\)? (i.e., what do we substitute for \(\alpha\)?)

\ldots we need a different type tag for \(d\)!

Idea

*We need to be able to substitute different types for type variables.*

\[
d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha
\]

⇒ *introduced by the let construct (hence the name)*
Typing Rules

- an identifier lookup

\[
(\forall \alpha_1, \ldots, \alpha_k. \tau) \sqsubseteq \tau'
\]

\[\pi \vdash x : \tau' \quad \text{for} \ (x : \forall \alpha_1, \ldots, \alpha_k. \tau) \in \pi
\]

\[\text{for} \ (\forall \alpha_1, \ldots, \alpha_k. \tau) \sqsubseteq \tau' \text{ if } [\tau_i/\alpha_i]_{\tau} = \tau' \text{ for some } \tau_1, \ldots, \tau_k.
\]

- $\lambda$-abstraction and application

\[
\pi \cup \{x : \tau\} \vdash E : \tau' \quad \pi \vdash E_1 : \tau \to \tau' \quad \pi \vdash E_2 : \tau
\]

\[
\pi \vdash \lambda x. E : \tau \to \tau'
\]

\[
\pi \vdash (E_1 \ E_2) : \tau'
\]

- let-abstraction

\[
\pi \vdash E_1 : \tau' \quad \pi \cup \{x : \text{Clos } \pi \ \tau'\} \vdash E_2 : \tau
\]

\[
\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau
\]

where $\text{Clos } \pi \ \tau = \forall \alpha_1, \ldots, \alpha_k. \tau$ where $\alpha_i \in \text{FV} (\tau) - \text{FV} (\pi)$. 

David Toman (University of Waterloo)
Some properties

- substitutions preserve well-formedness of terms:

**Theorem**

If \( \pi \vdash E : \tau \) then \( \sigma(\pi) \vdash E : \sigma(\tau) \) for all substitutions \( \sigma \).

- we lost unicity of typing... but there is a “substitute”:

**Definition (Principal Type)**

\( \tau \) is a principal type for \( E \) for \( \pi \) if

1. \( \pi \vdash E : \tau \), and
2. whenever \( \pi \vdash E : \tau' \) then there is a substitution \( \sigma \) such that \( \sigma(\tau) = \tau' \).

**Theorem**

There is unique principal type for every well-formed \( E \) (and \( \pi \)).
Some properties

- substitutions preserve well-formedness of terms:

**Theorem**

\[ \pi \vdash E : \tau \quad \text{then} \quad \sigma(\pi) \vdash E : \sigma(\tau) \quad \text{for all substitutions} \ \sigma. \]

- we lost *unicity of typing* ... but there is a “substitute”:

**Definition (Principal Type)**

\( \tau \) is a principal type for \( E \) for \( \pi \) if

1. \( \pi \vdash E : \tau \), and
2. whenever \( \pi \vdash E : \tau' \) then there is a substitution \( \sigma \) such that \( \sigma(\tau) = \tau' \).

**Theorem**

There is unique principal type for every well-formed \( E \) (and \( \pi \)).
Some properties

- substitutions preserve well-formedness of terms:

Theorem

If $\pi \vdash E : \tau$ then $\sigma(\pi) \vdash E : \sigma(\tau)$ for all substitutions $\sigma$.

- we lost unicity of typing . . . but there is a “substitute”:

Definition (Principal Type)

$\tau$ is a principal type for $E$ for $\pi$ if

1. $\pi \vdash E : \tau$, and
2. whenever $\pi \vdash E : \tau'$ then there is a substitution $\sigma$ such that $\sigma(\tau) = \tau'$.

Theorem

There is unique principal type for every well-formed $E$ (and $\pi$).
Typing Relation and Constraints

- We want to construct a principal type (if one exists)
- Example: an application

\[ \pi \vdash E_1 : \tau_1, \sigma_1 \quad \pi \vdash E_2 : \tau_2, \sigma_2 \]

\[ \pi \vdash (E_1 E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1 \]

How do we generate \( \tau_3 \)?

1. define an equation \( \tau_1 = \sigma_1 \tau_2 \rightarrow \alpha \) where \( \alpha \) is “fresh name”
2. if there is a solution \( \sigma_1 \), then the application is well formed
3. otherwise it cannot be well typed
4. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)

- to make this work we need to “collect” the substitutions on the fly
Typing Relation and Constraints

• We want to construct a *principal type* (if one exists)

• Example: an application

\[
\begin{align*}
\pi \vdash E_1 : \tau_1, \sigma_1 & \quad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2 \\
\pi \vdash (E_1 \ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1 
\end{align*}
\]

How do we generate \( \tau_3 \)?

1. define an equation “\( \tau_1 = \sigma \cdot \tau_2 \rightarrow \alpha \)” where \( \alpha \) is “fresh name”
2. if there is a solution, \( \sigma \), then the application is well formed
   \( \Rightarrow \) otherwise it cannot be well typed
3. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)

• to make this work we need to “collect” the substitutions on the fly
Typing Relation and Constraints

- We want to construct a *principal type* (if one exists)
- Example: an application

\[
\pi \vdash E_1 : \tau_1, \sigma_1 \quad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2
\]

\[
\pi \vdash (E_1 \ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1
\]

How do we generate \( \tau_3 \)?

1. define a *equation* “\( \tau_1 = \sigma_1 \tau_2 \rightarrow \alpha \)” where \( \alpha \) is “fresh name”
2. if there is a solution, \( \sigma \), then the application is well formed
   \( \Rightarrow \) otherwise it cannot be well typed
3. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)

- to make this work we need to “collect” the substitutions on the fly
Typing Relation and Constraints

- We want to construct a principal type (if one exists)
- Example: an application

\[ \pi \vdash E_1 : \tau_1, \sigma_1 \quad \sigma_1 \pi \vdash E_2 : \tau_2 \sigma_2 \]
\[ \pi \vdash (E_1 \ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1 \]

How do we generate \( \tau_3 \)?

1. define a equation “\( \tau_1 = \sigma_1 \tau_2 \rightarrow \alpha \)” where \( \alpha \) is “fresh name”
2. if there is a solution, \( \sigma \), then the application is well formed
\[ \Rightarrow \text{otherwise it cannot be well typed} \]
3. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)
- to make this work we need to “collect” the substitutions on the fly
Typing Relation and Constraints

- We want to construct a principal type (if one exists)
- Example: an application

\[ \pi \vdash E_1 : \tau_1, \sigma_1 \quad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2 \]
\[ \pi \vdash (E_1 \ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1 \]

How do we generate \( \tau_3 \)?

1. define a equation \( \tau_1 = \sigma_1 \tau_2 \rightarrow \alpha \) where \( \alpha \) is “fresh name”
2. if there is a solution, \( \sigma \), then the application is well formed
   \[ \Rightarrow \] otherwise it cannot be well typed
3. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)

• to make this work we need to “collect” the substitutions on the fly
Typing Relation and Constraints

- We want to construct a principal type (if one exists)
- Example: an application

\[ \pi \vdash E_1 : \tau_1, \sigma_1 \quad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2 \]

\[ \pi \vdash (E_1 \ E_2) : \tau_3, \sigma_2 \circ \sigma_1 \]

How do we generate \( \tau_3 \)?
1. define a equation “\( \tau_1 = \sigma_1 \tau_2 \rightarrow \alpha \)” where \( \alpha \) is “fresh name”
2. if there is a solution, \( \sigma \), then the application is well formed
   \[ \Rightarrow \] otherwise it cannot be well typed
3. for \( \sigma \) the most general solution we set \( \tau_3 = \sigma(\alpha) \)

- to make this work we need to “collect” the substitutions on the fly
How do we Solve the Constraints?

- tupe terms $\tau ::= i \mid \alpha \mid \tau \rightarrow \tau'$
- we use the unification algorithm [Robinson’65]

\[
\begin{align*}
\text{fun} & \quad \text{mgu} \quad i & i & = \{\} \\
& | \quad \text{mgu} \quad \alpha & \tau & = \{[\tau/\alpha]\} \quad \text{if } \alpha \notin FV(\tau) \\
& | \quad \text{mgu} \quad \tau & \alpha & = \{[\tau/\alpha]\} \quad \text{if } \alpha \notin FV(\tau) \\
& | \quad \text{mgu} \quad \tau_1 \rightarrow \tau'_1 & \tau_2 \rightarrow \tau'_2 & = (\text{mgu} \ \tau_1 \ \tau_2) \circ (\text{mgu} \ \tau'_1 \ \tau'_2) \\
& | \quad \text{mgu} \quad _ & _ & = \text{fail}
\end{align*}
\]

Theorem

$mgu(\tau_1, \tau_2)$ terminates and, if a substitution is returned, it is the most general unifier of $\tau_1$ and $\tau_2$. 
How do we Solve the Constraints?

- tupe terms $\tau ::= \iota \mid \alpha_i \mid \tau \rightarrow \tau'$
- we use the unification algorithm [Robinson’65]

<table>
<thead>
<tr>
<th>fun</th>
<th>mgu</th>
<th>$\iota$</th>
<th>$\iota$</th>
<th>${}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgu</td>
<td>$\alpha$</td>
<td>$\tau$</td>
<td>${[\tau/\alpha]}$ if $\alpha \notin \text{FV}(\tau)$</td>
<td></td>
</tr>
<tr>
<td>mgu</td>
<td>$\tau$</td>
<td>$\alpha$</td>
<td>${[\tau/\alpha]}$ if $\alpha \notin \text{FV}(\tau)$</td>
<td></td>
</tr>
<tr>
<td>mgu</td>
<td>$\tau_1 \rightarrow \tau'_1$</td>
<td>$\tau_2 \rightarrow \tau'_2$</td>
<td>$(\text{mgu } \tau_1 \tau_2) \circ (\text{mgu } \tau'_1 \tau'_2)$</td>
<td></td>
</tr>
<tr>
<td>mgu</td>
<td>_</td>
<td>_</td>
<td>fail</td>
<td></td>
</tr>
</tbody>
</table>

**Theorem**

$mgu(\tau_1, \tau_2)$ terminates and, if a substitution is returned, it is the **most general unifier** of $\tau_1$ and $\tau_2$. 
Algorithm W

- identifiers

\[(x : \forall \alpha_1, \ldots, \alpha_k. \tau) \in \pi \]

\[\pi \vdash x : [\beta_i/\alpha_i]\tau, \{\} \quad \text{where } \beta_i \text{ are fresh}\]

- \(\lambda\)-abstractions

\[\pi \cup \{x : \alpha\} \vdash E : \tau, \sigma \]

\[\pi \vdash \lambda x. E : (\sigma \alpha) \rightarrow \tau, \sigma \quad \text{where } \alpha \text{ is fresh}\]

- let-abstractions

How would a "recursive" let be defined?

\[\pi \vdash E_1 : \tau_1, \sigma_1 \]

\[\sigma_1 \cup \{x : \text{Clos}(\sigma_1 \pi) \tau_1\} \vdash E_2 : \tau_2, \sigma_2 \]

\[\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau_2, \sigma_2 \circ \sigma_1 \]

- and applications

\[\pi \vdash E_1 : \tau_1, \sigma_1 \quad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2 \]

\[\pi \vdash (E_1 E_2) : \sigma \alpha, \sigma \circ \sigma_2 \circ \sigma_1 \quad \text{for } \sigma = (\text{mgu}(\sigma_2 \tau_1)(\tau_2 \rightarrow \alpha))\]
Algorithm W

- identifiers

\[(x : \forall \alpha_1, \ldots, \alpha_k.\tau) \in \pi \]

\[\pi \vdash x : [\beta_i/\alpha_i]\tau, {}\]

where \(\beta_i\) are fresh

- \(\lambda\)-abstractions

\[\pi \cup \{x : \alpha\} \vdash E : \tau, \sigma\]

\[\pi \vdash \lambda x. E : (\sigma \alpha) \rightarrow \tau, \sigma\]

where \(\alpha\) is fresh

- let-abstractions

How would a "recursive" let be defined?

\[\pi \vdash E_1 : \tau_1, \sigma_1\]

\[(\sigma_1 \pi) \cup \{x : \text{Clos} (\sigma_1 \pi) \tau_1\} \vdash E_2 : \tau_2, \sigma_2\]

\[\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau_2, \sigma_2 \circ \sigma_1\]

- and applications

\[\pi \vdash E_1 : \tau_1, \sigma_1\]

\[(\sigma_1 \pi) \vdash E_2 : \tau_2, \sigma_2\]

\[\pi \vdash (E_1 E_2) : \sigma \alpha, \sigma \circ \sigma_2 \circ \sigma_1\]

for \(\sigma = (\text{mgu} (\sigma_2 \tau_1) (\tau_2 \rightarrow \alpha))\)
Algorithm W

- identifiers

\[(x : \forall \alpha_1, \ldots, \alpha_k. \tau) \in \pi\]
\[\pi \vdash x : [\beta_i/\alpha_i]\tau, \{\}\]

- **\(\lambda\)**-abstractions

\[\pi \cup \{x : \alpha\} \vdash E : \tau, \sigma\]
\[\pi \vdash \lambda x. E : (\sigma \alpha) \rightarrow \tau, \sigma\]  

- **let**-abstractions

\[\pi \vdash E_1 : \tau_1, \sigma_1\]
\[(\sigma_1 \pi) \cup \{x : \text{Clos} (\sigma_1 \pi) \tau_1\} \vdash E_2 : \tau_2, \sigma_2\]
\[\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau_2, \sigma_2 \circ \sigma_1\]

- and applications

\[\pi \vdash E_1 : \tau_1, \sigma_1\]
\[(\sigma_1 \pi) \vdash E_2 : \tau_2, \sigma_2\]
\[\pi \vdash (E_1 \ E_2) : \sigma \alpha, \sigma \circ \sigma_2 \circ \sigma_1\]

for \(\sigma = \text{mgu} (\sigma_2 \tau_1) (\tau_2 \rightarrow \alpha)\)
Algorithm W

- identifiers

\[
(x : \forall \alpha_1, \ldots, \alpha_k. \tau) \in \pi \\
\pi \vdash x : [\beta_i/\alpha_i] \tau, \{}
\]

where $\beta_i$ are fresh

- $\lambda$-abstractions

\[
\pi \cup \{x : \alpha\} \vdash E : \tau, \sigma \\
\pi \vdash \lambda x. E : (\sigma \alpha) \rightarrow \tau, \sigma
\]

where $\alpha$ is fresh

- let-abstractions

\[
\pi \vdash E_1 : \tau_1, \sigma_1 \\
(\sigma_1 \pi) \cup \{x : Clos(\sigma_1 \pi) \tau_1\} \vdash E_2 : \tau_2, \sigma_2
\]

\[
\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau_2, \sigma_2 \circ \sigma_1
\]

- and applications

\[
\pi \vdash E_1 : \tau_1, \sigma_1 \\
(\sigma_1 \pi) \vdash E_2 : \tau_2, \sigma_2
\]

\[
\pi \vdash (E_1 E_2) : \sigma \alpha, \sigma \circ \sigma_2 \circ \sigma_1
\]

for $\sigma = \langle \text{mgu}(\sigma_2 \tau_1)(\tau_2 \rightarrow \alpha) \rangle$

How would a “recursive” let be defined?
Algorithm W

- **identifiers**

\[
(x : \forall \alpha_1, \ldots, \alpha_k. \tau) \in \pi \quad \frac{\pi \vdash x : [\beta_i/\alpha_i]\tau, \{\}}{\pi \vdash x : \tau, \{\}} \quad \text{where } \beta_i \text{ are fresh}
\]

- **\(\lambda\)-abstractions**

\[
\pi \cup \{x : \alpha\} \vdash E : \tau, \sigma \quad \frac{\pi \vdash \lambda x. E : (\sigma \alpha) \rightarrow \tau, \sigma}{\pi \vdash \lambda x. E : \tau, \sigma} \quad \text{where } \alpha \text{ is fresh}
\]

- **let-abstractions**

\[
\pi \vdash E_1 : \tau_1, \sigma_1 \quad (\sigma_1 \pi) \cup \{x : \text{Clos}(\sigma_1 \pi) \tau_1\} \vdash E_2 : \tau_2, \sigma_2
\]

\[
\pi \vdash \text{let } x = E_1 \text{ in } E_2 : \tau_2, \sigma_2 \circ \sigma_1
\]

- **and applications**

\[
\pi \vdash E_1 : \tau_1, \sigma_1 \quad (\sigma_1 \pi) \vdash E_2 : \tau_2, \sigma_2
\]

\[
\pi \vdash (E_1 \ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1
\]

for \(\sigma = (\text{mgu} (\sigma_2 \tau_1) (\tau_2 \rightarrow \alpha))\)
Summary

- Polymorphic types convenient for code reuse
  
  - Let-polymorphism restricts the $\forall \alpha$. to the “top-level”
  
  - Type inference (reconstruction) algorithms and used.
    $\Rightarrow$ we need to be careful about sideeffects
    value restriction for type generalizations in SML
  
- Questions:
  1. How do recursive constructs interact with W?
  2. How do we add built-in operators? constants?
  3. What about disjunctive types (datatypes)?
  4. What about record types (hard!)
  5. Why don’t we use universal ($\forall \alpha.$) types in $\lambda$-abstraction?
Summary

- Polymorphic types convenient for code reuse
- Let-polymorphism restricts the $\forall \alpha$. to the “top-level”
- Type inference (reconstruction) algorithms and used.
  $\Rightarrow$ we need to be careful about sideeffects
  value restriction for type generalizations in SML
- Questions:
  1. How do recursive constructs interact with $W$?
  2. How do we add built-in operators? constants?
  3. What about disjunctive types (datatypes)?
  4. What about record types (hard!)
  5. Why don’t we use universal ($\forall \alpha$.) types in $\lambda$-abstraction?
Summary

- Polymorphic types convenient for code reuse
- Let-polymorphism restricts the $\forall \alpha$. to the “top-level”
- Type inference (reconstruction) algorithms and used.
  $\Rightarrow$ we need to be careful about sideeffects
  value restriction for type generalizations in SML

Questions:

1. How do recursive constructs interact with $W$?
2. How do we add built-in operators? constants?
3. What about disjunctive types (datatypes)?
4. What about record types (hard!)
5. Why don’t we use universal ($\forall \alpha.$) types in $\lambda$-abstraction?
Summary

- Polymorphic types convenient for code reuse
- Let-polymorphism restricts the \( \forall \alpha \) to the “top-level”
- Type inference (reconstruction) algorithms and used.
  \[ \Rightarrow \] we need to be careful about side effects
  value restriction for type generalizations in SML

Questions:
- 1. How do recursive constructs interact with \( W \)?
- 2. How do we add built-in operators? constants?
- 3. What about disjunctive types (data types)?
- 4. What about record types (hard!)
- 5. Why don’t we use universal (\( \forall \alpha \)) types in \( \lambda \)-abstraction?