Polymorphic Types and Type Inference

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

Type Variables and Substitutions

Idea

What should be a type annotation for "universal" functions?

$\Rightarrow$ *... we add variables for types.*

- Syntax of type annotations:

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

$\Rightarrow \alpha, \beta, \ldots$ are *type variables*

- How are the type variables used? substitutions $[\tau'/\alpha]\tau$!

Type Variables and Substitutions

Idea

What should be a type annotation for "universal" functions?

$\Rightarrow$ *...we add* *variables* *for types.*

- Syntax of type annotations:

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

  $\Rightarrow \alpha, \beta, \ldots$ are *type variables*

- How are the type variables used? substitutions $[\tau'/\alpha]\tau$

Type Variables and Substitutions

Idea

What should be a type annotation for "universal" functions?

$\Rightarrow$ . . . *we add variables for types.*

- Syntax of type annotations:

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

$\Rightarrow \alpha, \beta, \ldots$ are *type variables*

- How are the type variables used? substitutions

Type Variables and Substitutions

Idea

What should be a type annotation for "universal" functions?

$\Rightarrow$ *. . . we add variables for types.*

- Syntax of type annotations:

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

$\Rightarrow \alpha, \beta, \ldots$ are *type variables*

- How are the type variables used? substitutions $[\tau'/\alpha]\tau$!

Type Variables and Substitutions

Idea

What should be a type annotation for "universal" functions?

$\Rightarrow$ *...we add variables for types.*

- Syntax of type annotations:

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

$\Rightarrow \alpha, \beta, \ldots$ are *type variables*

- How are the type variables used? substitutions $[\tau'/\alpha]\tau$!

What does it Mean?

- Consider an expression

$$\lambda x : \alpha . \lambda y : \beta . (x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
    1. well typed for all substitutions for α and β (*universal reading*).
    2. well typed for some substitution for α, β (*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

$$\lambda x : \alpha . \lambda y : \beta . (x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

  . . . and this one is not!

  2. well typed for some substiturion for α, β (*existential reading*).

  . . . and this one is for $[\beta \to \beta / \alpha]$

Idea

We use type variables and substitutions "as general as possible" to infer/reconstruct type annotations.

What does it Mean?

- Consider an expression

$$\lambda x : \alpha . \lambda y : \beta . (x \; (x \; y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

     ... and this one is not!

  2. well typed for some substiturion for α, β (*existential reading*).

     ... and this one is for $[\beta \to \beta/\alpha]$

Idea

We use type variables and substitutions "as general as possible" to infer/reconstruct type annotations.

What does it Mean?

- Consider an expression

$$\lambda x:\alpha.\lambda y:\beta.(x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

     ...and this one is not!
  2. well typed for some substiturion for α, β (*existential reading*).

     ...and this one is for $[\beta \to \beta/\alpha]$ or [*int* $\to$ *int*/α, *int*/β]

Idea

We use type variables and substitutions "as general as possible" to infer/reconstruct type annotations.

What does it Mean?

- Consider an expression

$$\lambda x : \alpha . \lambda y : \beta . (x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

     . . . and this one is not!
  2. well typed for some substiturion for α, β (*existential reading*).

     . . . and this one is for $[\beta \rightarrow \beta/\alpha]$ or $[int \rightarrow int/\alpha, int/\beta]$

Idea

We use type variables and substitutions "as general as possible" to infer/reconstruct type annotations.

What does it Mean?

- Consider an expression

$$\lambda x\!:\!\alpha\,.\lambda y\!:\!\beta\,.(x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

     . . . and this one is not!
  2. well typed for some substiturion for α, β (*existential reading*).

     . . . and this one is for $[\beta \to \beta/\alpha]$ or $[int \to int/\alpha, int/\beta]$

Idea

We use type variables and substitutions "as general as possible" to infer/reconstruct type annotations.

What does it Mean?

- Consider an expression

$$\lambda x:\alpha.\lambda y:\beta.(x\ (x\ y))$$

$\Rightarrow$ is the expression well-formed (type-able)?

- two points of view:
  1. well typed for all substitutions for α and β (*universal reading*).

     ...and this one is not!

  2. well typed for some substiturion for α, β (*existential reading*).

     ...and this one is for $[\beta \to \beta/\alpha]$ or $[\mathit{int} \to \mathit{int}/\alpha, \mathit{int}/\beta]$

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:

$$\begin{aligned}\textbf{let}\ d = \lambda f : \alpha \to \alpha . \lambda x : \alpha . (f\ (f\ x))\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x : int . x + 1)\ 2\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x : str . x \hat{}\ x)\ \text{"}foo\text{"}\ \textbf{in} \ldots\end{aligned}$$

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

… we need a different type

Idea

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

$$d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha$$

$\Rightarrow$ *introduced by the let construct (hence the name)*

Let-Polymorphism

This is not quite good enough:

- consider the following expression:

$$\begin{aligned}\textbf{let}\ d = \lambda f{:}\alpha \to \alpha.\lambda x{:}\alpha.(f\ (f\ x))\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x{:}int.x + 1)\ 2\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x{:}str.x\hat{\ }x)\ "foo"\ \textbf{in}\ \ldots\end{aligned}$$

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

. . . we need a different type tag for d!

Idea

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

$$d : \forall\alpha : (\alpha \to \alpha) \to \alpha \to \alpha$$

$\Rightarrow$ *introduced by the let construct (hence the name)*

Let-Polymorphism

This is not quite good enough:

- consider the following expression:

$$\begin{aligned} \textbf{let}\ d = \lambda f{:}\alpha \to \alpha . \lambda x{:}\alpha . (f\ (f\ x))\ \textbf{in} & \\ \textbf{let}\ a = d\ (\lambda x{:}int . x + 1)\ 2\ \textbf{in} & \\ \textbf{let}\ a = d\ (\lambda x{:}str . x \,\hat{}\, x)\ \text{"}foo\text{"}\ \textbf{in} \ldots & \end{aligned}$$

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

...we need a different type tag for d!

Idea

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

$$d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha$$

$\Rightarrow$ *introduced by the **let** construct (hence the name)*

Let-Polymorphism

This is not quite good enough:

- consider the following expression:

$$\begin{aligned}\textbf{let}\ d = \lambda f : \alpha \to \alpha . \lambda x : \alpha . (f\ (f\ x))\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x : int . x + 1)\ 2\ \textbf{in}\\ \textbf{let}\ a = d\ (\lambda x : str . x \hat{}\ x)\ \text{"}foo\text{"}\ \textbf{in} \ldots\end{aligned}$$

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

…we need a different type tag for d!

Idea

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

$$d : \forall \alpha : (\alpha \to \alpha) \to \alpha \to \alpha$$

$\Rightarrow$ *introduced by the **let** construct (hence the name)*

Typing Rules

- an identifier lookup

$$\frac{(\forall \alpha_1, \ldots, \alpha_k.\tau) \sqsubseteq \tau'}{\pi \vdash x : \tau'} \quad \textit{for } (x : \forall \alpha_1, \ldots, \alpha_k.\tau\,) \in \pi$$

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

- λ-abstraction and application

$$\frac{\pi \uplus \{x : \tau\} \vdash E : \tau'}{\pi \vdash \lambda x.E : \tau \to \tau'} \qquad \frac{\pi \vdash E_1 : \tau \to \tau' \qquad \pi \vdash E_2 : \tau}{\pi \vdash (E_1\ E_2) : \tau'}$$

- let-abstraction

$$\frac{\pi \vdash E_1 : \tau' \qquad \pi \uplus \{x : \mathit{Clos}\ \pi\ \tau'\,\} \vdash E_2 : \tau}{\pi \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau}$$

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

Some properties

- substitutions preserve well-formedness of terms:

Theorem

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

Some properties

- substitutions preserve well-formedness of terms:

Theorem

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

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

Definition (Principal Type)

τ is a principal type for E for π if

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

Theorem

There is unique principal type for every well-formed E (and π).

Some properties

- substitutions preserve well-formedness of terms:

Theorem

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

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

Definition (Principal Type)

τ is a principal type for E for π if

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

Theorem

There is unique principal type for every well-formed E (and π).

Typing Relation and Constraints

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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1\pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

  How do we generate τ_3?

  1. define a *equation* “$\sigma_2\tau_1 = \tau_2 \to \alpha$” where α is “fresh name”
  2. if there is a solution, σ, then the application is well formed

     $\to$ otherwise it cannot be well typed
  3. for σ 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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

How do we generate τ_3?

1. define a *equation* “$\tau_1 = \sigma_1 \tau_2 \to \alpha$” where α is “fresh name”
2. if there is a solution, σ, then the application is well formed

   $\Rightarrow$ otherwise it cannot be well typed
3. for σ 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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1 \pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

How do we generate τ_3?

1. define a *equation* "$\tau_1 = \sigma_1 \tau_2 \to \alpha$" where α is "fresh name"
2. if there is a solution, σ, then the application is well formed

   $\Rightarrow$ otherwise it cannot be well typed
3. for σ 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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1\pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

  How do we generate τ_3?

  1. define a *equation* "$\tau_1 = \sigma_1\tau_2 \to \alpha$" where α is "fresh name"
  2. if there is a solution, σ, then the application is well formed

     $\Rightarrow$ otherwise it cannot be well typed
  3. for σ 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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1\pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

How do we generate τ_3?

1. define a *equation* "$\tau_1 = \sigma_1\tau_2 \to \alpha$" where α is "fresh name"
2. if there is a solution, σ, then the application is well formed
   $\Rightarrow$ otherwise it cannot be well typed
3. for σ 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

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad \sigma_1\pi \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \tau_3, \sigma \circ \sigma_2 \circ \sigma_1}$$

How do we generate τ_3?

1. define a *equation* "$\tau_1 = \sigma_1\tau_2 \to \alpha$" where α is "fresh name"
2. if there is a solution, σ, then the application is well formed
   $\Rightarrow$ otherwise it cannot be well typed
3. for σ 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 $\tau ::= \iota \mid \alpha_i \mid \tau \to \tau'$
- we use the unification algorithm [Robinson'65]

$$
\begin{array}{llccl}
\textbf{fun} & \textbf{mgu} & \iota & \iota & = \{\} \\
\mid & \textbf{mgu} & \alpha & \tau & = \{[\tau/\alpha]\} \;\; \textit{if } \alpha \notin FV(\tau) \\
\mid & \textbf{mgu} & \tau & \alpha & = \{[\tau/\alpha]\} \;\; \textit{if } \alpha \notin FV(\tau) \\
\mid & \textbf{mgu} & \tau_1 \to \tau_1' & \tau_2 \to \tau_2' & = (\textbf{mgu}\ \tau_1\ \tau_2) \circ (\textbf{mgu}\ \tau_1'\ \tau_2') \\
\mid & \textbf{mgu} & \_ & \_ & = \textbf{fail}
\end{array}
$$

Theorem

$\textbf{mgu}(\tau_1, \tau_2)$ terminates and, if a substitution is returned, it is the most general unifier of τ_1 and τ_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]

$$
\begin{array}{llccl}
\textbf{fun} & \textbf{mgu} & \iota & \iota & = \{\} \\
\mid & \textbf{mgu} & \alpha & \tau & = \{[\tau/\alpha]\} \;\; \textit{if } \alpha \notin FV(\tau) \\
\mid & \textbf{mgu} & \tau & \alpha & = \{[\tau/\alpha]\} \;\; \textit{if } \alpha \notin FV(\tau) \\
\mid & \textbf{mgu} & \tau_1 \rightarrow \tau_1' & \tau_2 \rightarrow \tau_2' & = (\textbf{mgu}\ \tau_1\ \tau_2) \circ (\textbf{mgu}\ \tau_1'\ \tau_2') \\
\mid & \textbf{mgu} & \_ & \_ & = \textbf{fail}
\end{array}
$$

Theorem

mgu(τ_1, τ_2) *terminates and, if a substitution is returned, it is the most general unifier of* τ_1 *and* τ_2.

Algorithm W

- identifiers

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

- λ-abstractions

$$\frac{\pi \uplus \{x : \alpha\} \vdash E : \tau, \sigma}{\pi \vdash \lambda x.E : (\sigma\ \alpha) \to \tau, \sigma} \quad \textit{where } \alpha \textit{ is fresh}$$

- **let**-abstractions How would a "recursive" let be defined?

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \uplus \{x : \mathit{Clos}\ (\sigma_1\ \pi)\ \tau_1\} \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau_2, \sigma_2 \circ \sigma_1}$$

- and applications

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1} \quad \textit{for } \sigma = (\textbf{mgu}\ (\sigma_2\ \tau_1)\ (\tau_2 \to \alpha))$$

Algorithm W

- identifiers

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

- λ-abstractions

$$\frac{\pi \uplus \{x : \alpha\} \vdash E : \tau, \sigma}{\pi \vdash \lambda x.E : (\sigma\ \alpha) \to \tau, \sigma} \quad \textit{where } \alpha \textit{ is fresh}$$

- **let**-abstractions

How would a "recursive" **let** be defined?

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \uplus \{x : \mathit{Clos}\ (\sigma_1\ \pi)\ \tau_1\} \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau_2, \sigma_2 \circ \sigma_1}$$

- and applications

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1} \quad \textit{for } \sigma = (\textbf{mgu}\ (\sigma_2\ \tau_1)\ (\tau_2 \to \alpha))$$

Algorithm W

- identifiers

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

- λ-abstractions

$$\frac{\pi \uplus \{x : \alpha\} \vdash E : \tau, \sigma}{\pi \vdash \lambda x.E : (\sigma\ \alpha) \to \tau, \sigma} \quad \textit{where } \alpha \textit{ is fresh}$$

- **let**-abstractions

How would a "recursive" **let** be defined?

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \uplus \{x : \mathit{Clos}\ (\sigma_1\ \pi)\ \tau_1\} \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau_2, \sigma_2 \circ \sigma_1}$$

- and applications

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1} \quad \textit{for } \sigma = (\textbf{mgu}\ (\sigma_2\ \tau_1)\ (\tau_2 \to \alpha))$$

Algorithm W

- identifiers

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

- λ-abstractions

$$\frac{\pi \uplus \{x : \alpha\} \vdash E : \tau, \sigma}{\pi \vdash \lambda x.E : (\sigma\ \alpha) \to \tau, \sigma} \quad \textit{where } \alpha \textit{ is fresh}$$

- **let**-abstractions

How would a "recursive" **let** be defined?

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \uplus \{x : \mathit{Clos}\ (\sigma_1\ \pi)\ \tau_1\} \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash \textbf{let}\ x = E_1\ \textbf{in}\ E_2 : \tau_2, \sigma_2 \circ \sigma_1}$$

- and applications

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1} \quad \textit{for } \sigma = (\textbf{mgu}\ (\sigma_2\ \tau_1)\ (\tau_2 \to \alpha))$$

Algorithm W

- identifiers

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

- λ-abstractions

$$\frac{\pi \uplus \{x : \alpha\} \vdash E : \tau, \sigma}{\pi \vdash \lambda x.E : (\sigma\ \alpha) \to \tau, \sigma} \quad \textit{where } \alpha \textit{ is fresh}$$

- **let**-abstractions How would a "recursive" **let** be defined?

$$\frac{\pi \vdash E_1 : \tau_1 , \sigma_1 \qquad (\sigma_1\ \pi) \uplus \{x : \mathit{Clos}\ (\sigma_1\ \pi)\ \tau_1\} \vdash E_2 : \tau_2 , \sigma_2}{\pi \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : \tau_2 , \sigma_2 \circ \sigma_1}$$

- and applications

$$\frac{\pi \vdash E_1 : \tau_1, \sigma_1 \qquad (\sigma_1\ \pi) \vdash E_2 : \tau_2, \sigma_2}{\pi \vdash (E_1\ E_2) : \sigma\alpha, \sigma \circ \sigma_2 \circ \sigma_1} \quad \textit{for } \sigma = (\mathbf{mgu}\ (\sigma_2\ \tau_1)\ (\tau_2 \to \alpha))$$

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 *resursive* constructs interact with W?
  2. How do we add *built-in* operators? constants?
  3. What about *disjunctive* types (`datatype`s)?
  4. What about *record* types (hard!)
  5. Why don't we use *universal* ($\forall\alpha.$) *types* in λ-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 *resursive* constructs interact with W?
  2. How do we add *built-in* operators? constants?
  3. What about *disjunctive* types (`datatype`s)?
  4. What about *record* types (hard!)
  5. Why don't we use *universal* ($\forall\alpha.$) *types* in λ-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 *resursive* constructs interact with W?
  2. How do we add *built-in* operators? constants?
  3. What about *disjunctive* types (`datatype`s)?
  4. What about *record* types (hard!)
  5. Why don't we use *universal* ($\forall\alpha.$) *types* in λ-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 *resursive* constructs interact with W?
  2. How do we add *built-in* operators? constants?
  3. What about *disjunctive* types (`datatype`s)?
  4. What about *record* types (hard!)
  5. Why don't we use *universal* ($\forall\alpha.$) *types* in λ-abstraction?