Tutorium 13: Compiler

Paul Brinkmeier

16. Februar 2021

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- Blatt 11 (Typinferenz und Let)
- Übersicht Compilerbau
- Syntaktische Analyse

Blatt 11

Aufgabe 2 — λ-Terme und ihre allgemeinsten Typen

Gegeben seien folgende λ-Terme:

$$
\begin{aligned}
t_1 &= \lambda z.\, z \\
t_2 &= \lambda f.\, \lambda x.\, f\ x \\
t_3 &= \lambda f.\, \lambda x.\, f\ (f\ x) \\
t_4 &= \lambda x.\, \lambda y.\, y\ (x\ y)
\end{aligned}
$$

Führen Sie für jeden dieser Terme eine Typinferenz durch.

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.\, b : \pi \to \rho}\textsc{Abs} \quad \rightsquigarrow \quad \begin{array}{c} \dfrac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\textsc{Abs} \\ \{\alpha_i = \alpha_j \to \alpha_k\} \end{array}$$

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\textsc{App} \quad \rightsquigarrow \quad \frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\begin{array}{c}\Gamma \vdash f\ x : \alpha_i \\ \{\alpha_j = \alpha_k \to \alpha_i\}\end{array}}\textsc{App}$$

Von Typisierungsregeln zu Typinferenz

Beim inferieren wird das Pattern-matching der Typen durch die Unifikation übernommen. Deswegen schreiben wir anstelle von konkreten Typen immer α_i und merken uns die Gleichungen für später:

$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}\textsc{Var} \quad \rightsquigarrow \quad \begin{array}{c}\dfrac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\textsc{Var} \\ \{\alpha_i = \alpha_j\}\end{array}$$

Algorithmus zur Typinferenz

- Stelle Typherleitungsbaum auf
  - In jedem Schritt werden neue Typvariablen α_i angelegt
  - Statt die Typen direkt im Baum einzutragen, werden Gleichungen in einem Constraint-System eingetragen
- Unifiziere Constraint-System zu einem Unifikator
  - Robinson-Algorithmus, im Grunde wie bei Prolog
  - I.d.R.: Allgemeinster Unifikator (mgu)

$$\frac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\textsc{Var}$$

Constraint:
$\{\alpha_i = \alpha_j\}$

$$\frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\Gamma \vdash f\ x : \alpha_i}\textsc{App}$$

Constraint:
$\{\alpha_j = \alpha_k \to \alpha_i\}$

$$\frac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\textsc{Abs}$$

Constraint:
$\{\alpha_i = \alpha_j \to \alpha_k\}$

Aufgabe 2 — t_2

$$\dfrac{\cdots}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}$$

$$C = \{ \qquad \}$$

Aufgabe 2 — t_2

$$\dfrac{\dfrac{\dots}{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}\textsc{Abs}}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}$$

$$C = \{ \qquad \}$$

Aufgabe 2 — t_2

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma \vdash f\ x : \alpha_5}\textsc{App}}{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}\textsc{Abs}}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}$$

$$\Gamma = f : \alpha_2, x : \alpha_4$$

$$C = \{ \qquad\qquad \}$$

Aufgabe 2 — t_2

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(f) = \alpha_2}{\Gamma \vdash f : \alpha_6}\textsc{Var} \qquad \dfrac{\Gamma(x) = \alpha_4}{\Gamma \vdash x : \alpha_7}\textsc{Var}}{\Gamma \vdash f\ x : \alpha_5}\textsc{App}}{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}\textsc{Abs}}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}$$

$$\Gamma = f : \alpha_2, x : \alpha_4$$

$$C = \{ \qquad \}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(f) = \alpha_2}{\Gamma \vdash f : \alpha_6}\,\textsc{Var} \qquad \dfrac{\Gamma(x) = \alpha_4}{\Gamma \vdash x : \alpha_7}\,\textsc{Var}}{\Gamma \vdash f\ x : \alpha_5}\,\textsc{App}}{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}\,\textsc{Abs}}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\,\textsc{Abs}$$

$$\Gamma = f : \alpha_2, x : \alpha_4$$

$$C = \{\alpha_1 = \alpha_2 \to \alpha_3,$$

$$\}$$

Aufgabe 2 — t_2

$$\dfrac{\dfrac{\dfrac{\Gamma(f) = \alpha_2}{\Gamma \vdash f : \alpha_6}\textsc{Var} \qquad \dfrac{\Gamma(x) = \alpha_4}{\Gamma \vdash x : \alpha_7}\textsc{Var}}{\Gamma \vdash f\ x : \alpha_5}\textsc{App}}{\dfrac{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}}\textsc{Abs}$$

$$\Gamma = f : \alpha_2, x : \alpha_4$$

$$C = \{\alpha_1 = \alpha_2 \to \alpha_3, \alpha_3 = \alpha_4 \to \alpha_5,$$

$$\}$$

Aufgabe 2 — t_2

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(f) = \alpha_2}{\Gamma \vdash f : \alpha_6}\textsc{Var} \qquad \dfrac{\Gamma(x) = \alpha_4}{\Gamma \vdash x : \alpha_7}\textsc{Var}}{\Gamma \vdash f\ x : \alpha_5}\textsc{App}}{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}\textsc{Abs}}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}$$

$$\Gamma = f : \alpha_2, x : \alpha_4$$

$$C = \{\alpha_1 = \alpha_2 \to \alpha_3, \alpha_3 = \alpha_4 \to \alpha_5,$$
$$\alpha_6 = \alpha_7 \to \alpha_5,$$
$$\}$$

Aufgabe 2 — t_2

$$
\dfrac{\dfrac{\dfrac{\Gamma(f) = \alpha_2}{\Gamma \vdash f : \alpha_6}\textsc{Var} \qquad \dfrac{\Gamma(x) = \alpha_4}{\Gamma \vdash x : \alpha_7}\textsc{Var}}{\Gamma \vdash f\ x : \alpha_5}\textsc{App}}{\dfrac{f : \alpha_2 \vdash \lambda x.\, f\ x : \alpha_3}{\vdash \lambda f.\, \lambda x.\, f\ x : \alpha_1}\textsc{Abs}}\textsc{Abs}
$$

$$
\begin{aligned}
\Gamma &= f : \alpha_2, x : \alpha_4 \\
C &= \{\alpha_1 = \alpha_2 \to \alpha_3, \alpha_3 = \alpha_4 \to \alpha_5, \\
&\qquad \alpha_6 = \alpha_7 \to \alpha_5, \\
&\qquad \alpha_6 = \alpha_2, \alpha_7 = \alpha_4\}
\end{aligned}
$$

Aufgabe 2 — t_2

$$
\begin{aligned}
C = \{&\alpha_1 = \alpha_2 \to \alpha_3, \alpha_3 = \alpha_4 \to \alpha_5,\\
&\alpha_6 = \alpha_7 \to \alpha_5,\\
&\alpha_6 = \alpha_2, \alpha_7 = \alpha_4\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{unify}(C) = \sigma = [&\alpha_1 \Rightarrow (\alpha_7 \to \alpha_5) \to \alpha_7 \to \alpha_5,\\
&\alpha_2 \Rightarrow \alpha_7 \to \alpha_5, \alpha_3 \Rightarrow \alpha_7 \to \alpha_5,\\
&\alpha_4 \Rightarrow \alpha_7, \alpha_6 \Rightarrow \alpha_7 \to \alpha_5]
\end{aligned}
$$

Also:

$$
\vdash \lambda f.\, \lambda x.\, f\ x : \sigma(\alpha_1) = (\alpha_7 \to \alpha_5) \to \alpha_7 \to \alpha_5
$$

Aufgabe 3 — Typabstraktion

In der Typabstraktion $ta(\tau, \Gamma)$ werden nicht alle freien Typvariablen von τ quantifiziert, sondern nur die, die nicht frei in den Typannahmen Γ vorkommen.

Überlegen sie anhand des λ-Terms $\lambda x.\, \texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x$, was passiert, wenn man diese Beschränkung aufhebt!

Let-Polymorphismus: Motivation

$$\lambda f.\, f\ f$$

- Diese Funktion verwendet f auf zwei Arten:
  - α: Rechte Seite.
  - $\alpha \to \beta$: Linke Seite, nimmt f als Argument und gibt es zurück.

Let-Polymorphismus: Motivation

$$\lambda f.\, f\ f$$

- Diese Funktion verwendet f auf zwei Arten:
  - α: Rechte Seite.
  - $\alpha \to \beta$: Linke Seite, nimmt f als Argument und gibt es zurück.
- Problem: α und $\alpha \to \beta$ sind nicht unifizierbar!
  - „occurs check“: α darf sich nicht selbst einsetzen.
- Idee: Bei jeder Verwendung eines polymorphen Typen erzeugen wir <u>neue Typvariablen</u>, um diese Beschränkung zu umgehen.
- Anders gesagt: Wir bauen uns ein f, was verschiedene Typen annehmen kann.

Typschemata und Instanziierung

- Idee: Bei jeder Verwendung eines polymorphen Typen erzeugen wir neue Typvariablen, um diese Beschränkung zu umgehen.
- Ein Typschema ist ein Typ, in dem manche Typvariablen allquantifiziert sind:

$$\phi = \forall \alpha_1. \dots \forall \alpha_n . \tau$$
$$\alpha_i \in FV(\tau)$$

- Typschemata kommen bei uns immer nur in Kontexten vor!
- Beispiele:
  - $\Gamma = f : \forall \alpha . \alpha \to \alpha$
  - $\Gamma' = g : \forall \alpha . \alpha \to \texttt{int} \to \alpha$

Typschemata und Instanziierung

- Ein Typschema kann zu verschiedenen Typen instanziiert werden.
- Allquantifizierte Typvariablen werden mit Typen belegt:

$$\forall\alpha.\alpha \to \alpha \succeq \mathsf{int} \to \mathsf{int} \;\; (\alpha \Leftarrow \mathsf{int})$$

$$\forall\alpha.\alpha \to \alpha \succeq \tau \to \tau \;\; (\alpha \Leftarrow \tau)$$

$$\forall\alpha.\alpha \to \alpha \not\succeq \tau \to \sigma \;\; (\tau \neq \sigma)$$

$$\forall\alpha.\alpha \to \alpha \not\succeq \tau \to (\tau \to \tau) \;\; (\tau \neq \tau \to \tau)$$

$$\forall\alpha.\alpha \to \alpha \succeq (\tau \to \tau) \to (\tau \to \tau) \;\; (\alpha \Leftarrow \tau \to \tau)$$

Instanziierung in Haskell

Typvariablen in Haskell sind <u>implizit</u> allquantifiziert:

```
                   (a -> b) -> [a] -> [b]
≈ forall {a} {b}. (a -> b) -> [a] -> [b]
```

Instanziierung in Haskell

Typvariablen in Haskell sind implizit allquantifiziert:

$$
\begin{aligned}
&\texttt{(a -> b) -> [a] -> [b]} \\
\approx\ &\texttt{forall \{a\} \{b\}. (a -> b) -> [a] -> [b]}
\end{aligned}
$$

GHCi kann uns diese auch zeigen:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> :set -fprint-explicit-foralls
Prelude> :t map
map :: forall {a} {b}. (a -> b) -> [a] -> [b]
```

Instanziierung in Haskell

Typvariablen in Haskell sind <u>implizit</u> allquantifiziert:

```
              (a -> b) -> [a] -> [b]
≈ forall {a} {b}. (a -> b) -> [a] -> [b]
```

Wenn wir Parameter in `map` einsetzen, werden `a` und `b` instanziiert:

```
Prelude> :t map not
map not :: [Bool] -> [Bool]
Prelude> :t map head
map head :: [[b]] -> [b]
Prelude> :t map sqrt
map sqrt :: [Float] -> [Float]
```

Let-Polymorphismus

Um Typschemata bei der Inferenz zu verwenden, müssen wir zunächst die Regel für Variablen anpassen:

$$\frac{\Gamma(x) = \phi \qquad \phi \succeq_{\textsf{frische } \alpha_i} \tau}{\Gamma \vdash x : \alpha_j}\textsc{Var}$$

Constraint: $\{\alpha_j = \tau\}$

- $\succeq_{\textsf{frische } \alpha_i}$ instanziiert ein Typschema mit α_i, die noch nicht im Baum vorkommen.
- Jetzt brauchen wir noch eine Möglichkeit, Typschemata zu erzeugen.

Let-Polymorphismus

Mit einen Let-Term wird ein Typschema eingeführt:

$$\frac{\Gamma \vdash t_1 : \alpha_i \qquad \tilde{\Gamma}, x : \tilde{\alpha}_i \vdash t_2 : \alpha_j}{\Gamma \vdash \texttt{let}\ \ x = t_1\ \ \texttt{in}\ \ t_2 : \alpha_k}\textsc{Let}$$

$$\begin{aligned} \sigma_{let} &= \texttt{unify}(C_{let}) \\ \tilde{\Gamma} &= \sigma_{let}(\Gamma) \\ \tilde{\alpha}_i &= ta(\sigma_{let}(\alpha_i), \tilde{\Gamma}) \end{aligned}$$

$ta(\tau, \Gamma)$ erzeugt ein Typschema, das die Typvariablen aus τ allquantifiziert, die nicht in Γ vorkommen.

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\textsc{Var}
      \qquad
      \dfrac{\Gamma'(x) = \text{int} \quad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\textsc{Var}
    }{\Gamma' \vdash f\ x : \alpha_4}\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{ \qquad \}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \quad = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad , \qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\Gamma'(f) = \alpha_3 \qquad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\,\textsc{Var}
      \qquad
      \dfrac{\Gamma'(x) = \mathsf{int} \qquad \mathsf{int} \succeq \mathsf{int}}{\Gamma' \vdash x : \alpha_6}\,\textsc{Var}
    }{\Gamma' \vdash f\ x : \alpha_4}\,\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\,{\color{red}\textsc{Abs}}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\,\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \mathsf{int} \\
\Gamma' &= x : \mathsf{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
& \qquad\qquad\qquad\qquad \}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \quad = x : \mathsf{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad\qquad\qquad , \qquad\quad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\,\textsc{Var}
      \qquad
      \dfrac{\Gamma'(x) = \text{int} \quad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\,\textsc{Var}
    }{\Gamma' \vdash f\ x : \alpha_4}\,{\color{red}\textsc{App}}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\,\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\,\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \qquad \qquad \qquad \}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \quad = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad \qquad , \qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\textsc{Var}
    \qquad
    \dfrac{\Gamma'(x) = \text{int} \quad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\textsc{Var}
  }{\Gamma' \vdash f\ x : \alpha_4}\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ g = \lambda f.\, f\ x\ \texttt{in}\ t : \alpha_1}\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \qquad \}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \quad = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad , \qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\textsc{Var}
    \qquad
    \dfrac{\Gamma'(x) = \text{int} \quad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\textcolor{red}{\textsc{Var}}
  }{\Gamma' \vdash f\ x : \alpha_4}\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\textsc{Abs}
  \qquad
  \dfrac{\ldots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \text{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \quad = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad , \qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\,\textsc{Var}
    \qquad
    \dfrac{\Gamma'(x) = \mathsf{int} \quad \mathsf{int} \succeq \mathsf{int}}{\Gamma' \vdash x : \alpha_6}\,\textsc{Var}
  }{\Gamma' \vdash f\ x : \alpha_4}\,\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\,\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\,\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \mathsf{int} \\
\Gamma' &= x : \mathsf{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \mathsf{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= [\alpha_2 \Rightarrow (\mathsf{int} \to \alpha_4) \to \alpha_4, \dots] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \qquad = x : \mathsf{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad\qquad\qquad\qquad , \qquad\quad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\begin{array}{c}\Gamma'(f) = \alpha_3 \\ \alpha_3 \succeq \alpha_3\end{array}}{\Gamma' \vdash f : \alpha_5}\textsc{Var}
    \qquad
    \dfrac{\begin{array}{c}\Gamma'(x) = \mathsf{int} \\ \mathsf{int} \succeq \mathsf{int}\end{array}}{\Gamma' \vdash x : \alpha_6}\textsc{Var}
  }{\Gamma' \vdash f\ x : \alpha_4}\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \mathsf{int} \\
\Gamma' &= x : \mathsf{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \mathsf{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= [\alpha_2 \Rightarrow (\mathsf{int} \to \alpha_4) \to \alpha_4, \dots] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma = x : \mathsf{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta(\qquad\qquad\qquad , \qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\Gamma'(f) = \alpha_3 \quad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\,\textsc{Var}
    \qquad
    \dfrac{\Gamma'(x) = \text{int} \quad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\,\textsc{Var}
  }{\Gamma' \vdash f\ x : \alpha_4}\,\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\,\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\,\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \text{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= [\alpha_2 \Rightarrow (\text{int} \to \alpha_4) \to \alpha_4, \dots] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta((\text{int} \to \alpha_4) \to \alpha_4, \qquad\qquad) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\dfrac{\Gamma'(f) = \alpha_3}{\alpha_3 \succeq \alpha_3}}{\Gamma' \vdash f : \alpha_5}\,\textsc{Var}
    \qquad
    \dfrac{\dfrac{\Gamma'(x) = \mathsf{int}}{\mathsf{int} \succeq \mathsf{int}}}{\Gamma' \vdash x : \alpha_6}\,\textsc{Var}
  }{\Gamma' \vdash f\ x : \alpha_4}\,\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\,\textsc{Abs}
  \qquad
  \dfrac{\dots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \mathtt{let}\ \ g = \lambda f.\, f\ x\ \ \mathtt{in}\ \ t : \alpha_1}\,\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= {\color{red} x : \mathsf{int}} \\
\Gamma' &= x : \mathsf{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \mathsf{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \mathtt{unify}(C_{let}) \\
&= [\alpha_2 \Rightarrow (\mathsf{int} \to \alpha_4) \to \alpha_4, \dots] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma = x : \mathsf{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta((\mathsf{int} \to \alpha_4) \to \alpha_4, x : \mathsf{int}) \\
&=
\end{aligned}
$$

Typabstraktion: Beispiel

$$
\dfrac{
  \dfrac{
    \dfrac{\dfrac{\Gamma'(f) = \alpha_3 \qquad \alpha_3 \succeq \alpha_3}{\Gamma' \vdash f : \alpha_5}\textsc{Var}
    \qquad
    \dfrac{\Gamma'(x) = \text{int} \qquad \text{int} \succeq \text{int}}{\Gamma' \vdash x : \alpha_6}\textsc{Var}}
    {\Gamma' \vdash f\ x : \alpha_4}\textsc{App}
  }{\Gamma \vdash \lambda f.\, f\ x : \alpha_2}\textsc{Abs}
  \qquad
  \dfrac{\ldots}{\tilde{\Gamma}, g : \tilde{\alpha}_2 \vdash t : \alpha_7}
}{\Gamma \vdash \texttt{let}\ \ g = \lambda f.\, f\ x\ \ \texttt{in}\ \ t : \alpha_1}\textsc{Let}
$$

$$
\begin{aligned}
\Gamma &= x : \text{int} \\
\Gamma' &= x : \text{int}, f : \alpha_3 \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_6 \to \alpha_4, \\
&\qquad \alpha_5 = \alpha_3, \alpha_6 = \text{int}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma_{let} &= \texttt{unify}(C_{let}) \\
&= [\alpha_2 \Leftarrow (\text{int} \to \alpha_4) \to \alpha_4, \ldots] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma = x : \text{int} \\
\tilde{\alpha}_2 &= ta(\sigma_{let}(\alpha_2), \tilde{\Gamma}) \\
&= ta((\text{int} \to \alpha_4) \to \alpha_4, x : \text{int}) \\
&= \forall \alpha_4.(\text{int} \to \alpha_4) \to \alpha_4
\end{aligned}
$$

Aufgabe 3 — Typabstraktion

$$
\dfrac{
\dfrac{
\dfrac{\begin{array}{c}\Gamma(x) = \alpha_2 \\ \alpha_2 \succeq \alpha_2\end{array}}{\Gamma \vdash x : \alpha_4}\,{\color{red}\textsc{Var}}
\qquad
\dfrac{\dots}{\tilde{\Gamma}, y : \tilde{\alpha}_4 \vdash y\ x : \alpha_5}\,\textsc{App}
}{\Gamma \vdash \texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x : \alpha_3}\,\textsc{Let}
}{\vdash \lambda x.\,\texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x : \alpha_1}\,\textsc{Abs}
$$

$$
\begin{aligned}
\Gamma &= x : \alpha_2 \\
\sigma_{let} &= [\alpha_4 \Rightarrow \alpha_2] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \\
\tilde{\alpha}_4 &= ta(\sigma_{let}(\alpha_4), \emptyset) = ta(\quad\ , \emptyset) = \forall \alpha_2 . \alpha_2 \\
&\neq ta(\alpha_2, x : \alpha_2) = \alpha_2
\end{aligned}
$$

Aufgabe 3 — Typabstraktion

$$\dfrac{\dfrac{\begin{array}{c}\Gamma(x) = \alpha_2 \\ \alpha_2 \succeq \alpha_2\end{array}}{\Gamma \vdash x : \alpha_4}\textsc{Var} \qquad \dfrac{\ldots}{\tilde{\Gamma}, y : \tilde{\alpha}_4 \vdash y\ x : \alpha_5}\textsc{App}}{\dfrac{\Gamma \vdash \texttt{let}\ y = x\ \texttt{in}\ y\ x : \alpha_3}{\vdash \lambda x.\,\texttt{let}\ y = x\ \texttt{in}\ y\ x : \alpha_1}\textsc{Abs}}\textsc{Let}$$

$$\begin{aligned}
\Gamma &= x : \alpha_2 \\
\sigma_{let} &= [\alpha_4 \Rightarrow \alpha_2] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma \\
\tilde{\alpha}_4 &= ta(\sigma_{let}(\alpha_4), \emptyset) = ta(\quad, \emptyset) = \forall \alpha_2.\alpha_2 \\
&\neq ta(\alpha_2, x : \alpha_2) = \alpha_2
\end{aligned}$$

Aufgabe 3 — Typabstraktion

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = \alpha_2 \qquad \alpha_2 \succeq \alpha_2}{\Gamma \vdash x : \alpha_4}\textsc{Var} \qquad \dfrac{\dots}{\tilde{\Gamma}, y : \tilde{\alpha}_4 \vdash y\ x : \alpha_5}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x : \alpha_3}\textsc{Let}}{\vdash \lambda x.\,\texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x : \alpha_1}\textsc{Abs}$$

$$\begin{aligned}
\Gamma &= x : \alpha_2 \\
\sigma_{let} &= [\alpha_4 \Leftarrow \alpha_2] \\
\tilde{\Gamma} &= \sigma_{let}(\Gamma) = \Gamma \\
\tilde{\alpha}_4 &= ta(\sigma_{let}(\alpha_4), \emptyset) = ta(\alpha_2, \emptyset) = \forall \alpha_2.\alpha_2 \\
&\neq ta(\alpha_2, x : \alpha_2) = \alpha_2
\end{aligned}$$

Aufgabe 3 — Typabstraktion

$$f = \lambda x.\,\texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x$$

Falsch:

$$\dfrac{\dfrac{(...)(y) = \forall\alpha_2.\alpha_2 \qquad \forall\alpha_2.\alpha_2 \succeq \alpha_8}{... \vdash y : \alpha_6}\,\textsc{Var} \qquad \dfrac{(...)(x) = \alpha_2 \qquad \alpha_2 \succeq \alpha_2}{... \vdash x : \alpha_7}\,\textsc{Var}}{x : \alpha_2, y : \forall\alpha_2.\alpha_2 \vdash y\ x : \alpha_5}\,\textsc{App}$$

Typisierbar, ermöglicht aber kaputten Code, bspw. $f\ 42 \Rightarrow 42\ 42 \not\Rightarrow$

Aufgabe 3 — Typabstraktion

$$f = \lambda x.\,\texttt{let}\ \ y = x\ \ \texttt{in}\ \ y\ x$$

Falsch:

$$\dfrac{\dfrac{(\ldots)(y) = \forall\alpha_2.\alpha_2 \qquad \forall\alpha_2.\alpha_2 \succeq \alpha_8}{\ldots \vdash y : \alpha_6}\,\textsc{Var} \qquad \dfrac{(\ldots)(x) = \alpha_2 \qquad \alpha_2 \succeq \alpha_2}{\ldots \vdash x : \alpha_7}\,\textsc{Var}}{x : \alpha_2, y : \forall\alpha_2.\alpha_2 \vdash y\ x : \alpha_5}\,\textsc{App}$$

Typisierbar, ermöglicht aber kaputten Code, bspw. $f\ 42 \Rightarrow 42\ 42 \not\Rightarrow$

Richtig:

$$\dfrac{\dfrac{(\ldots)(y) = \alpha_2 \qquad \alpha_2 \succeq \alpha_2}{\ldots \vdash y : \alpha_6}\,\textsc{Var} \qquad \dfrac{(\ldots)(x) = \alpha_2 \qquad \alpha_2 \succeq \alpha_2}{\ldots \vdash x : \alpha_7}\,\textsc{Var}}{x : \alpha_2, y : \alpha_2 \vdash y\ x : \alpha_5}\,\textsc{App}$$

Nicht typisierbar, äquivalent zu $\lambda x.\,x\ x$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

Bestimmen Sie einen allgemeinsten Typ für den Ausdruck

$$\texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\,(k\ b\ c)$$

unter der Typannahme $\Gamma = a : \text{int}, b : \text{bool}, c : \text{char}$.

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\ (k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\ (k\ b\ c) : \alpha_1}$$

$$\Gamma = a : \text{int}, b : \text{bool}, c : \text{char}$$

$$C_{let} =$$

$$\sigma_{let} = \texttt{unify}(C_{let}) =$$

$$C'_{let} =$$

$$\Gamma' =$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\,(k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\,(k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \text{int}, b : \text{bool}, c : \text{char} \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, ...\} \\
\sigma_{let} = \texttt{unify}(C_{let}) &= \\
C'_{let} &= \\
\Gamma' &=
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\ (k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\ (k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \mathrm{int}, b : \mathrm{bool}, c : \mathrm{char} \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \dots\} \\
\sigma_{let} = \texttt{unify}(C_{let}) &= [\alpha_2 \Rightarrow \alpha_6 \to \alpha_5 \to \alpha_6, \dots] \\
C'_{let} &= \{\alpha_2 = \alpha_6 \to \alpha_5 \to \alpha_6, \dots\} \\
\Gamma' &=
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\,(k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\,(k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \text{int}, b : \text{bool}, c : \text{char} \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \dots\} \\
\sigma_{let} = \texttt{unify}(C_{let}) &= [\alpha_2 \Diamond \alpha_6 \to \alpha_5 \to \alpha_6, \dots] \\
C'_{let} &= \{\alpha_2 = \alpha_6 \to \alpha_5 \to \alpha_6, \dots\} \\
\Gamma' &= \sigma_{let}(\Gamma), k : ta(\sigma_{let}(\alpha_2), \sigma_{let}(\Gamma))
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\ (k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\ (k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \text{int}, b : \text{bool}, c : \text{char} \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \dots\} \\
\sigma_{let} = \texttt{unify}(C_{let}) &= [\alpha_2 \Rightarrow \alpha_6 \to \alpha_5 \to \alpha_6, \dots] \\
C'_{let} &= \{\alpha_2 = \alpha_6 \to \alpha_5 \to \alpha_6, \dots\} \\
\Gamma' &= \sigma_{let}(\Gamma), k : ta(\sigma_{let}(\alpha_2), \sigma_{let}(\Gamma)) \\
&= \Gamma, k : ta(\alpha_6 \to \alpha_5 \to \alpha_6, \Gamma)
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\, (k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\, (k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \text{int}, b : \text{bool}, c : \text{char} \\
C_{let} &= \{\alpha_2 = \alpha_3 \to \alpha_4, \dots\} \\
\sigma_{let} = \texttt{unify}(C_{let}) &= [\alpha_2 \Rightarrow \alpha_6 \to \alpha_5 \to \alpha_6, \dots] \\
C'_{let} &= \{\alpha_2 = \alpha_6 \to \alpha_5 \to \alpha_6, \dots\} \\
\Gamma' &= \sigma_{let}(\Gamma), k : ta(\sigma_{let}(\alpha_2), \sigma_{let}(\Gamma)) \\
&= \Gamma, k : ta(\alpha_6 \to \alpha_5 \to \alpha_6, \Gamma) \\
&= \Gamma, k : \forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\ (k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\ (k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \mathsf{int}, b : \mathsf{bool}, c : \mathsf{char} \\
\Gamma' &= \Gamma, k : \forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 \\
\forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 &\succeq \mathsf{bool} \to \mathsf{char} \to \mathsf{bool} \\
\forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 &\succeq \mathsf{int} \to \mathsf{bool} \to \mathsf{int} \\
C &=
\end{aligned}$$

Aufgabe 4 — Typinferenz, Let-Polymorphismus

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \lambda x.\, \lambda y.\, x : \alpha_2}\textsc{Abs} \qquad \dfrac{\dots}{\Gamma' \vdash k\ a\,(k\ b\ c) : \alpha_7}\textsc{App}}{\Gamma \vdash \texttt{let}\ \ k = \lambda x.\, \lambda y.\, x\ \ \texttt{in}\ \ k\ a\,(k\ b\ c) : \alpha_1}$$

$$\begin{aligned}
\Gamma &= a : \mathsf{int}, b : \mathsf{bool}, c : \mathsf{char} \\
\Gamma' &= \Gamma, k : \forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 \\
\forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 &\succeq \mathsf{bool} \to \mathsf{char} \to \mathsf{bool} \\
\forall\alpha_6.\forall\alpha_5.\alpha_6 \to \alpha_5 \to \alpha_6 &\succeq \mathsf{int} \to \mathsf{bool} \to \mathsf{int} \\
C &= C'_{let} \cup C_{body} \cup \{\alpha_1 = \alpha_7\}
\end{aligned}$$

Einführung in Compilerbau

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung

Compiler in ProPa

- Ein bisschen...
  - Lexikalische Analyse (Tokenisierung)
  - Syntaktische Analyse (Parsen)
  - Semantische Analyse (Optimierung)
  - Codegenerierung
- Klausur:
  - SLL(k)-Form beweisen
  - Rekursiven Abstiegsparser schreiben/vervollständigen
  - First/Follow-Mengen berechnen
  - Java-Bytecode

Syntaktische Analyse (18WS)

$$\begin{aligned} \mathit{SGML} &\rightarrow \ \texttt{<}\ \texttt{id}\ \texttt{>}\ \ \mathit{Children}\ \ \texttt{<}\ \texttt{/}\ \texttt{>} \\ \mathit{Children} &\rightarrow \ \epsilon \mid \mathit{SGML}\ \ \mathit{Children} \end{aligned}$$

$$\{\texttt{<id><id></><id></></>},\ \ \texttt{<id></>},\ \ ...\} \in G$$

Syntaktische Analyse (18WS)

$$SGML \rightarrow \texttt{< id >}\ Children\ \texttt{< / >}$$
$$Children \rightarrow \epsilon \mid SGML\ \ Children$$

$$\{\texttt{<id><id></><id></></>},\ \texttt{<id></>},\ ...\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in $\mathrm{SLL}(1)$-Form ist (3P.).

Syntaktische Analyse (18WS)

$$
\begin{aligned}
\mathit{SGML} &\rightarrow \texttt{ < id > } \mathit{Children} \texttt{ < / > } \\
\mathit{Children} &\rightarrow \epsilon \mid \mathit{SGML}\ \ \mathit{Children}
\end{aligned}
$$

$$\{\texttt{<id><id></><id></></>},\ \texttt{<id></>},\ \ldots\} \in G$$

- Begründen Sie formal, dass die obige Grammatik nicht in $\mathrm{SLL}(1)$-Form ist (3P.).
- Entwickeln Sie für [eine linksfaktorisierte Version der obigen Grammatik] einen rekursiven Abstiegsparser (16P.).

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Java-Bytecode (16SS)

Übersetzen Sie folgenden Java-Programmausschnitt in Java-Bytecode (10P.):

```
if (((a < b) || !((a < c) || (c < b))) && !(c < 0)) {
    c = b + a;
}
```

Hinweise:

- Codeerzeugung für bedingte Sprünge: Folien 447ff.
- Um eine Bedingung der Form `!cond` zu übersetzen, reicht es, `cond` zu übersetzen und die Sprungziele anzupassen.

Compiler

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher (= billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.

Compiler: Motivation

- Maschine(-nmodell) versteht i.d.R. eingeschränkten Instruktionssatz
  - Es gibt/gab zwar auch mal CISC-Maschinen, heute ist sind aber RISC(-ähnliche) Prozessoren am weitesten verbreitet
  - Gründe: RISC-Prozessoren sind wesentlich einfacher (= billiger) zu bauen
- Programme in Maschinensprache sind i.d.R. für Menschen nicht einfach zu Schreiben.
- Also: Erfinde einfacher zu Schreibende (≈ mächtigere) Sprache, die dann in die Sprache der Maschine übersetzt wird.
- Diesen Übersetzungsschritt sollte optimalerweise ein Programm erledigen, da wir sonst auch einfach direkt Maschinensprache-Programme schreiben können.

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go → X86
  - Java, Clojure, Kotlin → Java-Bytecode
  - TypeScript → JavaScript
  - Python → Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu

Compiler

- Übersetzer für formale Sprachen nennt man Compiler
- Beispiele:
  - C, Haskell, Rust, Go $\rightarrow$ X86
  - Java, Clojure, Kotlin $\rightarrow$ Java-Bytecode
  - TypeScript $\rightarrow$ JavaScript
  - Python $\rightarrow$ Python-AST
- Interpreter kann man auch als Compiler kategorisieren, sie zählen aber i.A. nicht dazu
- Single-pass vs. Multi-pass
  - Single-pass: Eingabe wird einmal gelesen, Ausgabe währenddessen erzeugt (ältere Compiler)
  - Multi-pass: Eingabe wird in Zwischenschritten in verschiedene Repräsentationen umgewandelt
    - Quellsprache, Tokens, AST, Zwischensprache, Zielsprache

Lexikalische Analyse

```
int x1 = 123;
print("123");
```

```
int, id[x1], assign,
intlit[123], semi,
id[print], lp,
stringlit["123"], ...
```

- Lexikalische Analyse (Tokenisierung) verarbeitet eine Zeichensequenz in eine Liste von Tokens.
- Tokens sind Zeichengruppen, denen eine Semantik innewohnt:
  - `int` — Typ einer Ganzzahl
  - = — Zuweisungsoperator
  - `x1` — Variablen- oder Methodenname
  - 123 — Literal einer Ganzzahl
  - `"123"` — String-Literal
  - etc.
- Lösbar mit regulären Ausdrücken, Automaten

Syntaktische Analyse

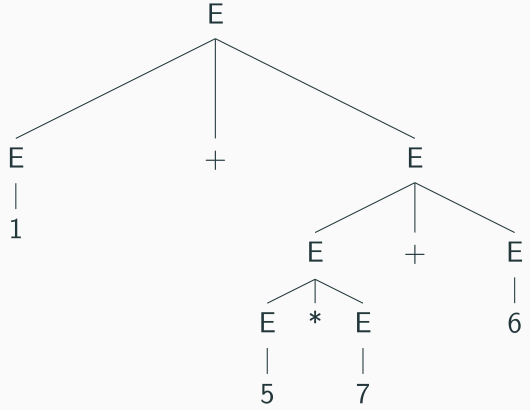
- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
    - Blockstruktur von Programmen
    - Baumstruktur von HTML-Dateien
    - Header, Inhalt-Struktur von Mails
    - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.

Syntaktische Analyse

- Syntaktische Analyse stellt die unterliegende Struktur der bisher linear gelesenen Eingabe fest:
  - Blockstruktur von Programmen
  - Baumstruktur von HTML-Dateien
  - Header, Inhalt-Struktur von Mails
  - Verschachtelte mathematische Ausdrücke
- Syntaktische Analyse ist das größte Compiler-Thema in PP.
- Übliche Vorgehensweise (in PP):
  - Grammatik G erfinden
  - ggf. G in einfache Form G' bringen
  - rekursiven Abstiegsparser für G' implementieren
- Alternativ: Parser-Kombinatoren, Yacc, etc.

Beispiel: Mathematische Ausdrücke

$1 + 5 * 7 + 6$



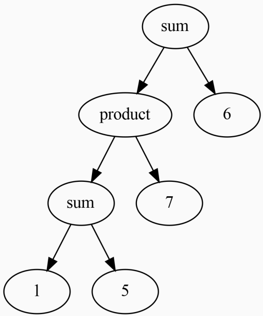
- Zu beachten: Punkt-vor-Strich (Präzedenz), Klammerung, etc.
- Nicht mehr mit regulären Ausdrücken lösbar
- „Offensichtliche“ Grammatik oft nicht einfach zu Parsen

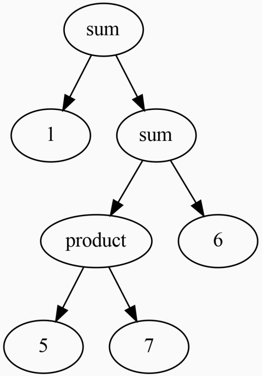
Beispiel: Mathematische Ausdrücke

$$E \rightarrow \ \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$

Beispiel: Mathematische Ausdrücke

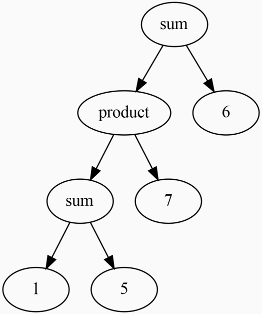
$$E \rightarrow \ \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$

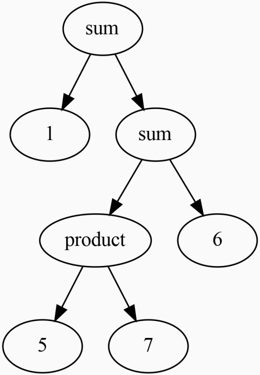




Beispiel: Mathematische Ausdrücke

$$E \rightarrow \ \texttt{num} \mid \texttt{(}\ E\ \texttt{)} \mid E\ \texttt{+}\ E \mid E\ \texttt{*}\ E$$





- Ableitungsbaum nicht eindeutig $\rightsquigarrow$ schlecht
- Ableitungsbaum garantiert nicht Punkt-vor-Strich $\rightsquigarrow$ schlecht

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$
\begin{aligned}
E &\to E \texttt{ + } T \mid E \texttt{ - } T \mid T \\
T &\to T \texttt{ * } F \mid T \texttt{ / } F \mid F \\
F &\to \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

Präzedenz, Linksfaktorisierung

Wie zeichnen sich „gute“ Grammatiken aus?

Operatorpräzedenz schon in Grammatik definiert:

$$
\begin{aligned}
E &\rightarrow E \texttt{ + } T \mid E \texttt{ - } T \mid T \\
T &\rightarrow T \texttt{ * } F \mid T \texttt{ / } F \mid F \\
F &\rightarrow \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

Vermeidung von Linksrekursion (Linksfaktorisierung):

$$
\begin{aligned}
E &\rightarrow T \; \mathit{EList} \\
\mathit{EList} &\rightarrow \epsilon \mid \texttt{+ } T \; \mathit{EList} \mid \texttt{- } T \; \mathit{EList} \\
T &\rightarrow F \; \mathit{TList} \\
\mathit{TList} &\rightarrow \epsilon \mid \texttt{* } F \; \mathit{TList} \mid \texttt{/ } F \; \mathit{TList} \\
F &\rightarrow \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\leadsto$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \leadsto$ weiter mit ϕ

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \ \epsilon \mid \texttt{+} \ \ T \ \ \mathit{EList} \mid \texttt{-} \ \ T \ \ \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- $IM_1(\mathit{EList} \rightarrow \ \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{)}, \texttt{\#}\}$
- $IM_1(\mathit{EList} \rightarrow \ \texttt{+} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{+} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{+}\}$
- $IM_1(\mathit{EList} \rightarrow \ \texttt{-} \ \ T \ \ \mathit{EList}) = \mathrm{First}_1(\texttt{-} \ T \ \mathit{EList} \ \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{-}\}$

First-/Followmenge, Indizmenge

$$\mathit{EList} \rightarrow \; \epsilon \mid \texttt{+} \;\; T \;\; \mathit{EList} \mid \texttt{-} \;\; T \;\; \mathit{EList}$$

Wie können wir bspw. bei *EList* entscheiden, welche Produktion anzuwenden ist?

- $\rightsquigarrow$ definiere Indizmenge $IM_k(A \rightarrow \alpha) = \mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$
- Wenn nächste k Token in $IM_k(\mathit{EList} \rightarrow \phi) \rightsquigarrow$ weiter mit ϕ
- $IM_1(\mathit{EList} \rightarrow \; \epsilon) = \mathrm{First}_1(\epsilon \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{)}, \texttt{\#}\}$
- $IM_1(\mathit{EList} \rightarrow \; \texttt{+} \;\; T \;\; \mathit{EList}) = \mathrm{First}_1(\texttt{+} \; T \; \mathit{EList} \; \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{+}\}$
- $IM_1(\mathit{EList} \rightarrow \; \texttt{-} \;\; T \;\; \mathit{EList}) = \mathrm{First}_1(\texttt{-} \; T \; \mathit{EList} \; \mathrm{Follow}_1(\mathit{EList})) = \{\texttt{-}\}$
- $\mathrm{First}_k(A)$: Menge an möglichen ersten k Token in A
- $\mathrm{Follow}_k(A)$: Menge an möglichen ersten k Token nach A

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form
$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

SLL-Kriterium

Grammatik ist in $\mathrm{SLL}(k)$-Form

$:\Leftrightarrow \forall A \to \alpha, A \to \beta \in P : IM_k(A \to \alpha) \cap IM_k(A \to \beta) = \emptyset$

- $\mathrm{SLL}(k)$: Bei jedem Nichtterminal muss die zu wählende Produktion an den nächsten k Token wählbar sein.
- Nichtterminale mit nur einer Produktion sind hier irrelevant
- Schwierig daran: Follow-Mengen berechnen

$$
\begin{aligned}
E &\to E \texttt{ + } T \mid E \texttt{ - } T \mid T \\
T &\to T \texttt{ * } F \mid T \texttt{ / } F \mid F \\
F &\to \texttt{num} \mid \texttt{(} E \texttt{)}
\end{aligned}
$$

- Begründet formal, dass obige Grammatik nicht $\mathrm{SLL}(1)$.
- Berechnet $\mathrm{Follow}_1(N)$ für $N \in \{E, T, F\}$.

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\to T \;\; \mathit{EList} \\
\mathit{EList} &\to \epsilon \mid \texttt{+} \;\; T \;\; \mathit{EList} \mid \texttt{-} \;\; T \;\; \mathit{EList} \\
T &\to F \;\; \mathit{TList} \\
\mathit{TList} &\to \epsilon \mid \texttt{*} \;\; F \;\; \mathit{TList} \mid \texttt{/} \;\; F \;\; \mathit{TList} \\
F &\to \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?

Rekursive Abstiegsparser

$$\begin{aligned} E &\rightarrow \; T \;\; EList \\ EList &\rightarrow \; \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\ T &\rightarrow \; F \;\; TList \\ TList &\rightarrow \; \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\ F &\rightarrow \; \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)} \end{aligned}$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?
- G ist jetzt einfach ausprogrammierbar:
  - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
  - `Token[k]`-Instanzattribut für k langen Lookahead
  - `expect(TokenType)`-Methode, um Token zu verarbeiten

Rekursive Abstiegsparser

$$
\begin{aligned}
E &\rightarrow \; T \;\; EList \\
EList &\rightarrow \; \epsilon \mid \texttt{+} \;\; T \;\; EList \mid \texttt{-} \;\; T \;\; EList \\
T &\rightarrow \; F \;\; TList \\
TList &\rightarrow \; \epsilon \mid \texttt{*} \;\; F \;\; TList \mid \texttt{/} \;\; F \;\; TList \\
F &\rightarrow \; \texttt{num} \mid \texttt{(} \;\; E \;\; \texttt{)}
\end{aligned}
$$

- Yay, unsere Grammatik hat jetzt $\mathrm{SLL}(1)$-Form!
- Aber was bringt das?
- *G* ist jetzt einfach ausprogrammierbar:
    - 1 Methode per Nichtterminal: `parseE()`, `parseEList()`, ...
    - `Token[k]`-Instanzattribut für *k* langen Lookahead
    - `expect(TokenType)`-Methode, um Token zu verarbeiten
- Vervollständigt `demos/java/exprparser/ExprParser.java`!

Semantische Analyse

- PP beschäftigt sich (bis auf Typinferenz) nur kurz mit semantischer Analyse
- Hier geht es um Optimierungen, Typchecks, etc.
- $\rightsquigarrow$ weiterführende (Master-)Vorlesungen am IPD

Ende

Letzte Folie

- Morgen, Mi, 17.02. um 14:00: Letzte Vorlesung
- Mi, 31.03. um 14:00: Fragestunde mit den Übungsleitern
    - Per Zoom, über den Vorlesungslink
- Tutoriumsfolien, -code, etc.: github.com/pbrinkmeier/pp-tut
- Fragen auch gerne an `pp-tut@pbrinkmeier.de` :)

Danke für's Kommen und eine gute Prüfungsphase!