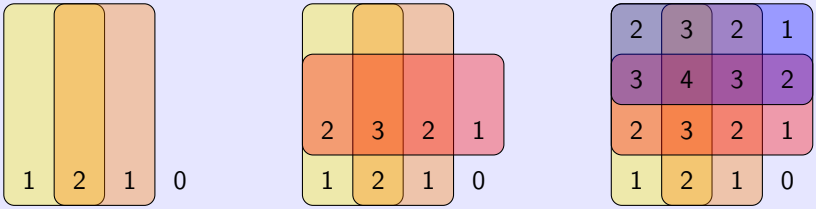
14. Knobelaufgabe #5

Mit zwei Rechtecken lassen sich $2^2 = 4$ „verschiedene“ Flächen konstruieren; mit drei Rechtecken $2^3 = 8$ und mit vier Rechtecken $2^4 = 16$. Zwei Flächen gelten als „gleich“, wenn sie von den gleichen Rechtecken überdeckt werden. (Die Zahlen in den Beispielen geben jeweils die Gesamtzahl der Überdeckungen an.)



Wieviele verschiedene Flächen lassen sich mit fünf Rechtecken erzeugen? Und mit sechs?

14. Motivation

- Zur Erinnerung:

  let $s = 4711 + 815$ ***in*** $s * s$

  berechnet den Flächeninhalt eines Quadrats der Seitenlänge $4711 + 815$.
- Wir können das Programm verallgemeinern, indem wir von der Seitenlänge *abstrahieren*: aus $s * s$ wird eine Funktion in s.

  let *area* $(s : Nat) : Nat = s * s$

  - *area* ist der Name der Funktion,
  - s vom Typ *Nat* ist der *formale Parameter* und
  - $s * s$ vom Typ *Nat* heißt *Rumpf* der Funktion.

14. Motivation

- Um den Flächeninhalt für eine gegebene Seitenlänge zu berechnen, wenden wir die Funktion an:

$$\mathit{area}\ (4711 + 815)$$

$4711 + 815$ ist das Argument oder der *aktuelle Parameter* der Funktion.

- Der Funktionsaufruf wird ausgerechnet, indem der formale Parameter an den aktuellen Parameter gebunden wird und in dieser Umgebung der Rumpf ausgewertet wird.

$$\mathit{area}\ (4711 + 815) \quad \cong \quad \textbf{\textit{let}}\ s = 4711 + 815\ \textbf{\textit{in}}\ s * s$$

14. Motivation

- Der Vorteil einer Funktionsdefinition ist, dass die Funktion mehrfach angewendet werden kann:

$$\mathit{area}\ (4711) + \mathit{area}\ (815)$$

- Ohne Funktionen im Repertoire müssten wir formulieren:

$$(\textbf{\textit{let}}\ s = 4711\ \textbf{\textit{in}}\ s * s) + (\textbf{\textit{let}}\ s = 815\ \textbf{\textit{in}}\ s * s)$$

☞ Der Ausdruck wird verdoppelt, nicht nur die Rechnung!

- Funktionsdefinitionen sind Definitionen und können wie diese in ***in***-Ausdrücken verwendet werden.

$$\begin{array}{l}\textbf{\textit{let}}\ \mathit{area}\ (s : \mathit{Nat}) : \mathit{Nat} = s * s \\ \textbf{\textit{in}}\ \ \mathit{area}\ (4711) + \mathit{area}\ (815)\end{array}$$

Sichtbarkeit: area ist im ***in***-Ausdruck sichtbar; der formale Parameter *s* hingegen nur im Funktionsrumpf.

14. Abstrakte Syntax

$f \in \mathsf{Id}$

$d ::= \cdots$	*Deklarationen:*
$\quad \mid$ ***let*** $f\ (x : t_1) : t_2 = e$	Funktionsdefinition
$e ::= \cdots$	*Funktionsausdrücke:*
$\quad \mid\ f\ (e)$	Funktionsapplikation

☞ t_1 ist der Argumenttyp von f und t_2 der Ergebnistyp.

☞ Funktionsapplikation ist der vornehme Name für Funktionsaufruf oder Funktionsanwendung.

14. Statische Semantik

Eine Funktion mit den Argumenttyp t_1 und dem Ergebnistyp t_2 erhält den Typ $t_1 \to t_2$.
Lies: t_1 nach t_2.

$t ::= \cdots$	*Typen:*
$\mid t_1 \to t_2$	Funktionstyp

Typregeln:

$$\frac{\Sigma, \{x_1 \mapsto t_1\} \vdash e_2 : t_2}{\Sigma \vdash (\textbf{\textit{let}}\ f\ (x_1 : t_1) : t_2 = e_2) : \{f \mapsto t_1 \to t_2\}}$$

$$\frac{\Sigma(f) = t_1 \to t_2 \qquad \Sigma \vdash e_1 : t_1}{\Sigma \vdash f\ (e_1) : t_2} \qquad f \in dom\ \Sigma$$

☞ Der Rumpf einer Funktion wird in der um den formalen Parameter erweiterten Signatur getypt (‘,’ ist der Kommaoperator).

14. Dynamische Semantik

- Was ist der Wert einer Funktion?
- Können wir Funktionen überhaupt auswerten?
- Man könnte eine Funktion tabellieren: für jedes Argument wird der Funktionswert ausgerechnet. Beispiel:

$$\textbf{\textit{let}}\ \mathit{not}\ (a : \mathit{Bool}) : \mathit{Bool} = \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ \mathit{false}\ \textbf{\textit{else}}\ \mathit{true}$$

wird ausgewertet zu

$$\{\mathit{false} \mapsto \mathit{true}, \mathit{true} \mapsto \mathit{false}\}$$

☞ Nicht machbar, wenn das Argument eine natürliche Zahl ist.

- *Idee:* die Auswertung einer Funktion wird verzögert oder "eingefroren", bis der aktuelle Parameter bekannt ist.

14. Verzögerte Auswertung

Was passiert, wenn der Funktionsrumpf freie Bezeichner enthält?

$$\dfrac{\dfrac{\varnothing \vdash 2 \Downarrow 2}{\varnothing \vdash \textbf{\textit{let}}\ d = 2 \Downarrow \{d \mapsto 2\}} \qquad \dfrac{}{\{d \mapsto 2\} \vdash \textbf{\textit{let}}\ \mathit{next}\,(n) = n + d \Downarrow ?}}{\varnothing \vdash \textbf{\textit{let}}\ d = 2\ \textbf{\textit{let}}\ \mathit{next}\,(n) = n + d \Downarrow ?}$$

☞ Wenn wir die Auswertung “einfrieren” und später fortsetzen (“auftauen”) wollen, müssen wir uns die Funktionsdefinition *und* die aktuelle Umgebung merken.

14. Dynamische Semantik

Der Bereich der Werte wird um *Funktionsabschlüsse* erweitert (engl. closures).

$\nu ::= \cdots$	*Werte:*
$\mid \langle \delta, x, e \rangle$	Funktionsabschluss

Für das obige Beispiel, die Funktion *next*, erhalten wir:

$$\langle \{ d \mapsto 2 \}, n, n + d \rangle$$

Die Funktionsdefinition, die Zuordnung von n zu $n + d$, wird festgehalten; die Umgebung $\{ d \mapsto 2 \}$ legt die Bedeutung des *freien* Bezeichners d fest.

14. Dynamische Semantik

Auswertungsregel:

$$\overline{\delta \vdash (\textbf{\textit{let}}\ f\ (x) = e) \Downarrow \{f \mapsto \langle \delta, x, e \rangle\}}$$

Eine Funktionsdefinition wertet zu einer Bindung aus, in der der Funktionsname an einen Funktionsabschluss gebunden ist.

Die Kombination aus Umgebung und Ausdruck, $\delta \vdash e$, kann als Konfiguration oder Zustand eines Rechners aufgefasst werden; die Auswertungsregeln legen die Arbeitsweise des Rechners fest; eine "closure" speichert im wesentlichen eine Konfiguration.

14. Dynamische Semantik

Auswertungsregel:

$$\frac{\delta(f) = \langle \delta', x_1, e \rangle \qquad \delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{x_1 \mapsto \nu_1\} \vdash e \Downarrow \nu}{\delta \vdash f\ (e_1) \Downarrow \nu}$$

☞ Jetzt da der aktuelle Parameter bekannt ist, kann die verzögerte Auswertung wiederaufgenommen werden.

14. Dynamische Semantik — Beispielrechnung

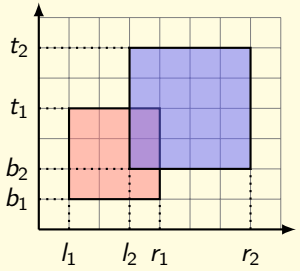
$$\dfrac{\dfrac{}{\varnothing \vdash \mathit{area}\ (s) = s * s \Downarrow \delta} \qquad \dfrac{\dfrac{}{\delta \vdash 4711 + 815 \Downarrow 5526} \quad \dfrac{}{\{s \mapsto 5526\} \vdash s * s \Downarrow 30536676}}{\delta \vdash \mathit{area}\ (4711 + 815) \Downarrow 30536676}}{\varnothing \vdash \textbf{\textit{let}}\ \mathit{area}\ (s) = s * s\ \textbf{\textit{in}}\ \mathit{area}\ (4711 + 815) \Downarrow 30536676}$$

wobei $\delta = \{\, \mathit{area} \mapsto \langle \varnothing, s, s * s \rangle \,\}$

☞ Der Bezeichner *area* wird an den Funktionsabschluss $\langle \varnothing, s, s * s \rangle$ gebunden.

14. Beispiel: Berechnung der Wohnfläche — da capo

Die Fläche einer Wohnung mit dem unten skizzierten Grundriss soll berechnet werden.



14. Beispiel: Berechnung der Wohnfläche — da capo

Alte Lösung:

$$
\begin{array}{l}
\textbf{let}\ l_1 = 1 \\
\textbf{let}\ r_1 = 4 \\
\textbf{let}\ b_1 = 1 \\
\textbf{let}\ t_1 = 4 \\
\\
\textbf{let}\ l_2 = 3 \\
\textbf{let}\ r_2 = 7 \\
\textbf{let}\ b_2 = 2 \\
\textbf{let}\ t_2 = 6 \\
\\
\textbf{let}\ s_1 = r_1 \dot{-} l_1 \\
\textbf{let}\ s_2 = r_2 \dot{-} l_2 \\
\textbf{let}\ w = \mathit{min}\ r_1\ r_2 \dot{-} \mathit{max}\ l_1\ \ l_2 \\
\textbf{let}\ h = \mathit{min}\ t_1\ t_2 \dot{-} \mathit{max}\ b_1\ b_2 \\
s_1 * s_1 + s_2 * s_2 \dot{-} w * h
\end{array}
$$

Neue Lösung:

$$
\begin{array}{l}
\textbf{let}\ \mathit{total\text{-}area}\ (l_1, r_1, b_1, t_1, l_2, r_2, b_2, t_2) : \mathit{Nat} = \\
\quad \textbf{let}\ s_1 = r_1 \dot{-} l_1 \\
\quad \textbf{let}\ s_2 = r_2 \dot{-} l_2 \\
\quad \textbf{let}\ w = \mathit{min}\ r_1\ r_2 \dot{-} \mathit{max}\ l_1\ \ l_2 \\
\quad \textbf{let}\ h = \mathit{min}\ t_1\ t_2 \dot{-} \mathit{max}\ b_1\ b_2 \\
\quad s_1 * s_1 + s_2 * s_2 \dot{-} w * h \\
\\
\textbf{let}\ \mathit{Wohnfläche} = \\
\quad \mathit{total\text{-}area}\ (1, 4, 1, 4, 3, 7, 2, 6)
\end{array}
$$

14. Demo

Mini⟩ *total-area* $(1, 4, 1, 4, 3, 7, 2, 6)$
23
Mini⟩ *total-area* $(3, 7, 2, 6, 1, 4, 1, 4)$
23
Mini⟩ *total-area* $(0, 4, 0, 4, 1, 2, 1, 2)$
16
Mini⟩ *total-area* $(0, 4, 0, 4, 4, 7, 4, 7)$
25

14. Vertiefung

Den Booleschen Verknüpfungen können wir jetzt einen Namen geben.

- *Negation*:

```
let not (a : Bool) : Bool = if a then false else true
```

- *Konjunktion*:

```
let and-also (a : Bool, b : Bool) : Bool = if a then b else false
```

- *Disjunktion*:

```
let or-else (a : Bool, b : Bool) : Bool = if a then true else b
```

☞ *and-also* und *or-else* haben zwei Parameter.

Funktionsabschluss? Wozu soll das gut sein?
Bachelorabschluss wär mir lieber.

Jedes Programm hat seinen Wert: der Abschluss $\langle \delta, x, e \rangle$ ist der Wert einer Funktion. Der Wert umfasst alles, was wir über eine Funktion wissen müssen: wie das Funktionsergebnis e in Abhängigkeit vom Funktionsargument x berechnet werden kann und welche Bedeutung die freien Variablen haben.

Müssen wir eigentlich Funktionen immer einen Namen geben? Das machen wir beim Rechnen mit natürlichen Zahlen ja auch nicht—jeder Zahl einen Namen geben, meine ich. Warum haben wir keine Ausdrücke, die zu Funktionen auswerten? Wenn Funktionen Werte sind ...

Gute Idee, Lisa!

15. Abstrakte Syntax

$e ::= \cdots$		*Funktionsausdrücke:*
	fun $(x : t) \to e$	Funktionsabstraktion
	$e\ e_1$	Funktionsapplikation

☞ ***fun*** $(x : t) \to e$ ist eine *anonyme* Funktion mit dem formalen Parameter x und dem Rumpf e.

☞ Funktionsaufrufe werden ebenfalls verallgemeinert: da Funktionen berechnet werden können, verallgemeinert sich $f\ e_1$ zu $e\ e_1$.

Aus der Schule ist man gewohnt, Applikationen zu klammern: $f\ (e)$ statt $f\ e$. In der konkreten Syntax sind Klammern nur nötig, wenn das Argument nicht „atomar“ ist: $f\ x$ und $f\ 4711$, aber $f\ (x + 4711)$. Später mehr dazu.

15. Statische Semantik

Typregeln:

$$\frac{\Sigma, \{x_1 \mapsto t_1\} \vdash e : t_2}{\Sigma \vdash (\textbf{\textit{fun}}\ (x_1 : t_1) \to e) : t_1 \to t_2}$$

$$\frac{\Sigma \vdash e : t_1 \to t_2 \qquad \Sigma \vdash e_1 : t_1}{\Sigma \vdash e\ e_1 : t_2}$$

15. Dynamische Semantik

Auswertungsregeln:

$$\frac{}{\delta \vdash (\textbf{\textit{fun}}\ x \to e) \Downarrow \langle \delta, x, e \rangle}$$

$$\frac{\delta \vdash e \Downarrow \langle \delta', x_1, e' \rangle \qquad \delta \vdash e_1 \Downarrow \nu_1 \qquad \delta', \{x_1 \mapsto \nu_1\} \vdash e' \Downarrow \nu'}{\delta \vdash e\ e_1 \Downarrow \nu'}$$

☞ Die Auswertung der Funktionsapplikation ist jetzt etwas komplizierter, da zusätzlich e ausgerechnet werden muss — vorher stand an dieser Stelle ein Bezeichner.

Die Auswertung einer Applikation vollzieht sich somit in drei Schritten:

1. die Funktion wird ausgerechnet;
2. das Argument wird ausgerechnet;
3. der Rumpf der Funktion wird ausgerechnet.

15. Vertiefung

- Auch mit Funktionen lässt sich vorzüglich rechnen.
- Die folgenden Definitionen setzen die Addition und die Multiplikation auf Funktionen fort.

$$\textbf{\textit{let}}\ add\ (f : Nat \to Nat, g : Nat \to Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to f\ x + g\ x$$

$$\textbf{\textit{let}}\ mul\ (f : Nat \to Nat, g : Nat \to Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to f\ x * g\ x$$

- *add* und *mul* arbeiten auf Funktionen des Typs $Nat \to Nat$.
- Mit Hilfe von *id* und *constant* lassen sich Funktionen des Typs $Nat \to Nat$ konstruieren.

$$\textbf{\textit{let}}\ id : Nat \to Nat = \textbf{\textit{fun}}\ (x : Nat) \to x$$

$$\textbf{\textit{let}}\ constant\ (n : Nat) : Nat \to Nat =$$
$$\quad \textbf{\textit{fun}}\ (x : Nat) \to n$$

15. Demo

```
Mini⟩ add (id, constant 4711)
val it : Nat → Nat
Mini⟩ (add (id, constant 4711)) 815
5526
Mini⟩ add (id, constant 4711) 815
5526
Mini⟩ let square = mul (id, id)
val square : Nat → Nat
Mini⟩ square 4711
22193521
Mini⟩ let twice (f : Nat → Nat) : Nat → Nat = fun x → f (f x)
val twice : (Nat → Nat) → (Nat → Nat)
Mini⟩ twice square 2
16
Mini⟩ twice (twice square) 2
65536
```

Wenn ich das richtig sehe, dann sind Funktionsdefinitionen gar nicht mehr notwendig: die Funktionsdefinition

$$\textbf{\textit{let}}\ f\ (x : t_1) : t_2 = e$$

entspricht exakt der Wertedefinition

$$\textbf{\textit{let}}\ f = \textbf{\textit{fun}}\ (x : t_1) \to e$$

Hmm, Du meinst wohl, anonyme Funktionen sind überflüssig. Statt

$$\textbf{\textit{fun}}\ (x : t_1) \to e$$

schreibe ich immer

$$\textbf{\textit{let}}\ f\ (x : t_1) : t_2 = e\ \textbf{\textit{in}}\ f$$

Also Harry, ich erkenne Dich gar nicht wieder:
28 Tastendrücke statt 17. Sonst bist Du doch so tippfaul.