17.11.05

Theorie der Programmierung I

(Mitschrift von Simon Schöling)

Bisher: $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$ ungetypt (kontextfrei)
Nachteil: Programmierfehler werden alle erst zur Laufzeit entdeckt („Steckenbleiben“)
Deshalb: Einführung eines statischen Typsystems (Typen werden zur Compilezeit überprüft). Zur Laufzeit gibt es kein „Steckenbleiben“ mehr, grobe Programmierfehler werden schon erkannt. ➔ Es werden **einfach getypte** Versionen der Sprachen $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$ definiert. Bezeichnung: $\mathcal{L}_0^t, \mathcal{L}_1^t, \mathcal{L}_2^t$

Was heißt einfach getypt?

- Man hat keine Polymorphie und kein Subtyping
- Man hat keine Typinferenz, d.h. Argument- und Reultattyp einer Funktion müssen stets vom Programmierer angegeben werden

Definition: Die Menge von Typen aller Typen τ ist definiert duch:

$\tau ::=$ unit | bool | int Basistypen
| $\tau_1 \rightarrow \tau_2$ Funktionstypen

konkrete Syntax „$\rightarrow$“ ist rechtsassoziativ*, d.h. $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ steht für $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$

* *links assoziativ würde bedeuten, dass Funktionen als Argument eine Funktion erwartet, ist eher selten, daher ohne Klammern bedeutet rechtsassoziativ.*

Die kontextfreie Syntax der Ausdrücke wird so verändert:

e ::= λid : τ.e_1 (statt λid. e_1)
e ::= rec id : τ.e_1 (statt rec id e_1)

Veränderung des syntaktischen Zuckers (siehe Zettel)

Bsp.: let rec exp (x : int) (y : int) : int = if y = 0 … in exp m n
steht für:
let rec exp : int$\rightarrow$ int$\rightarrow$int. λx : int. λy : int. if … in m n
steht für:
let exp = rec exp : int$\rightarrow$int$\rightarrow$int. λx : int. λy : int. if … in m n
Damit ist die kontextfreie Syntax der neuen Sprachen definiert.
jetzt: Präzise Definition der „Kontextbedingunen“
nämlich: jeder Name muss „bekannt“ sein, bevor er benutzt wird (d.h. das Gesamt-Programm muss abgeschlossen sein.)
Alle Ausdrücke müssen „wohlgetypt“ sein. Dazu Typregeln für Konstanten und Ausdrücke.

Typurteile für Konstanten: c : τ
Die gültigen Typurteile sind:

() :: unit	(UNIT)
b :: bool	(BOOL)
n :: int	(INT)
op :: int$\rightarrow$int$\rightarrow$int	(AOP), falls op arithmetischer Operator
op :: int$\rightarrow$int$\rightarrow$bool	(BOP), falls op Vergleichsoperator

Wann ist ein Ausdruck e wohlgetypt? z.B. + x 1
Dazu muss der Typ von x bekannt sein. Diese Information muss aus dem „Kontext“ des Ausdrucks kommen: Wenn + x 1 im Gültigkeitsbereich von „λx : int“ oder let x = e

(Ausdruck vom Typ int) steht, dann ist er wohlgetypt. ➔ Solche Informationen aus dem Kontext muss man sich in einer „Tabelle“ merken, um die Typüberprüfung durchführen zu können.

Definition: Eine Typumgebung Γ ist eine partielle endliche Funktion Γ: id → Type, d.h. eine Funktion, deren Definitionsbereich dom(Γ) endlich ist.

Schreibweise (für Beispiel): $[id_1: \tau_1, \ldots, id_n: \tau_n]$ steht für die Funktion Γ mit
$dom(\Gamma) = \{i_1, \ldots, id_n\}$; $\Gamma(id_i) = \tau_1$ für i = 1, …, n

Typurteile für Ausdrücke sind von der Form $\Gamma \triangleright e::\tau$ (lies: „e hat Typ τ bezüglich Γ)

Schreibweise: Seien A, B Mengen, f: $A \hookrightarrow B$ eine partielle Funktion und seien $a \in A$ und $b \in B$. Dann bezeichnet man f [b/a] die Funktion, die sich von f nur dadurch unterscheidet, dass sie an der Stelle a das Resultat b liefert.

Formal: f [b/a] ist die Funktion g: $A \hookrightarrow B$ mit $dom(g) = dom(f) \cup \{a\}$

$$g(a') = \begin{cases} b \text{ falls } a' = a \\ f(a') \text{ falls } a' \in dom(g) \setminus \{a\} \end{cases}$$

Speziell: Γ [τ/id] ist die „Tabelle“, die aus Γ entsteht, indem man id : τ hinzunimmt, wobei ein alter Eintrag für id überschrieben wird.

Die gültigen Typurteile erhält man mit den Regeln (siehe Zettel):

(CONST): Eine Konstante c hat in jeder Umgebung Γ ihren vorgegebenen Typ

(ID): Ein Name hat bezüglich Γ den Typ, der dort steht

(APP): Die linke Seite einer Applikation muss Funktionstyp $\tau \to \tau'$ haben, die rechte Seite den zugehörigen Agumenttyp τ. Das Ergebnis ist dann vom Typ τ'

(COND): Die Bedingung im if-then-else muss Typ bool haben, then- und else-Teil müssen den gleichen Typ haben

(LET): Um den Typ von let id = e_1 in e_2 zu bestimmen, geht man so vor: man bestimmt den Typ von e1 (in der ursprünglichen Umgebung Γ). Diesen Typ trägt man für den Namen id in die tabelle ein und überprüft dann e_2

(ABSTR): Um den Typ von λid : τ.e zu bestimmen trägt man τ if id in Γ ein und überprüft dann e. Wenn e den Typ τ' hat, ist λid : τ.e vom Funktionstyp $\tau \to \tau'$

(REC): In rec id : τ.e ist τ der „gewünschte Typ“ der rekursiven Funktion id. Also muss man überprüfen, ob e diesen gewünschten Typ hat, wobei man für die rekursiven Aufrufe von id schon den gewünschten Typ voraussetzen darf

Beispiel:

(1) [] ▷ let rec exp = rec : int→int→int. λx : int. λy : int if = y 0 then 1 else ∗ x (exp x (- y 1))
in exp 3 4 :: int
↳ (LET) (2) [] rec exp : int→int→int … :: int
↳ (REC) (3) [exp : int→int→int] ▷ λx : int. λy : int if … :: int
↳ 2x (ABSTR) (4) [x : int, y : int exp : int→int→int] ▷ if = y 0 … :: int
↳ (COND) (5) […] ▷ = y 0 :: bool
↳ (APP) (6) […] ▷ = y :: int→bool
↳ (APP) (7) […] ▷ = :: int→int→bool (ROP)
↳ (8) […] ▷ y :: int (ID)
↳ (9) […] ▷ 0 :: int
↳ (CONST) (10) 0 :: int (INT)
↳ (11) […] ▷ 1 :: int (CONST), (INT)
↳ (12) […] ▷ ∗ x (exp x (- y 1)) :: int
…