15.12.05

Theorie der Programmierung I

(Mitschrift von Lars Friedrich email@lars-friedrich-home.de)

$\mathcal{L}_i^{ti}$: keine Polymorphie ➔ wir müssen unterscheiden zwischen reinen Platzhaltern α und Typparametern α.

Definition: (a) Die Menge PType der **polymorphen Typen** π sei definiert durch:
$\pi ::= \tau \mid \forall\, \alpha_1, \ldots, \alpha_n.\tau$ $(n \geq 1)$
Intuition: Die ∀-quantifizierten Variablen von τ sind reine Platzhalter.
(b) Die (gewöhnlichen) Typen $\tau \in$ Type nennt man jetzt **monomorphe** Typen (beachte: Type ⊆ PType)
(c) Eine **Typumgebung** Γ ist jetzt eine endliche partielle Funktion Γ: Id ↪ PType
(d) Ein **Typurteil** ist eine Formel $\Gamma \triangleright e :: \tau$ (mit der neuen Definition von Typumgebungen Γ)
| | monomorpher Typ
| enthält polymorphe Typen

Idee: Neu-Namen können polymorph sein, und zwar nur die Namen, die mit „let" eingeführt wurden (so genannte **let-Polymorphie**).

Beispiele: let f = λx. x in …
Da α ➔ α (allgemeinster) Typ von λx. x ist, wird für f der „entsprechende" polymorphe Typ ∀α: α ➔ α eingetragen.
Intuition: Für jeden Typ α kann f den Typ α ➔ α annehmen, z. B. f: int ➔ int, f: bool ➔ bool, … ➔ wir brauchen Typregeln, in denen zum Ausdruck kommt:
- über welche Typvariablen man beim Eintrag in Γ ∀-Quantoren setzen darf
- wie man „Instanzen" eines polymorphen Typs bildet

Definition: (a) Die Menge free(π) aller in π frei vorkommenden Typvariablen ist definiert durch: $free(\tau) = tvar(\tau)$, $free(\forall \alpha_1, \ldots, \alpha_n.\tau) = tvar(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$
(b) Die Menge free(Γ) aller in Γ frei vorkommenden Typvariablen ist definiert durch: $free(\Gamma) = \cup_{id \in dom(\Gamma)} free(\Gamma(id))$
(c) **Der polymorphe Abschluss** eines Typs τ in der Typumgebung Γ ist definiert durch: $closure(\tau) = \forall \alpha_1, \ldots, \alpha_n.\tau$ wobei $\{ \alpha_1, \ldots, \alpha_n \} = tvar(\tau) \setminus free(\Gamma)$, d.h. die Typen aus τ, die nicht frei in Γ vorkommen

Die Sprache $\mathcal{L}_i^{ML}$ (i=0, …, 3) sind wie folgt definiert:
- die kontextfreie Syntax ist die von $\mathcal{L}_i^{ti}$
- die Typregeln entstehen aus denen von $\mathcal{L}_i^{ti}$, wobei man (LET) und (ID) durch zwei neue Regeln ersetzt:

(P-LET) $$\frac{\Gamma \triangleright e_1 :: \tau_1 \quad \Gamma[closure_\Gamma(\tau_1) / id] \triangleright e_2 :: \tau_2}{\Gamma \triangleright \text{let id} = e_1 \text{ in } e_2}$$

(P-ID) $\Gamma \triangleright id :: \tau\ [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ falls $id \in dom(\Gamma)$ und
| Instanz von
$\Gamma(id) = \forall \alpha_1, \ldots, \alpha_n.\tau$

Intuition: (zu (P-LET)) Jede „neue" Typvariable im Typ τ_1 von e_1 steht für einen beliebigen Typ, deshalb darf man einen ∀-Quantor darüber setzen. Aber: Eine „alte" Typvariable ($\in free(\Gamma)$) bezieht sich auf einen bereits bekannten Typ.

Beispiel: let rec exists p l = not(is_empty) && (p (hd l) || exists p (tl l)) in exists (λx. x mod 2 = 0) [1, 2, 3] && exists (λx. x) [false, true]

(1) let exists = rec exists λp. λl. not …
↳(2) (P-LET) rec exists … :: (α → bool) → α list → bool
↳(3) [exists: ∀α (α → bool) → α list → bool] ▷ exists … && … :: bool
↳(4) (REC) [exists: ∀α (α → bool) → α list → bool] ▷ λp. λl. … :: (α→bool)→α list→bool
↳(5) (ABSTR) [exists: …, p: α→bool] ▷ λl. … :: α list→bool
↳(6) (ABSTR) [exists: …, p: …, l: α list] ▷ not(is_empty l) && … :: bool
↳(7) (AND) […] ▷ … :: …
↳(8) (…) p (hd l) ||exists p (tl l) :: bool
↳(9) (OR) […] ▷ … :: …
↳(10) (…) […] ▷ exists p (tl l) :: bool
↳(11) (…) […] ▷ exists p :: α list→bool
↳(12) (APP) […] ▷ exists :: (α→bool)→α list→bool mit (P-ID)
↳(13) (…) […] ▷ p :: α→bool mit (P-ID)
↳(14) (…) […] ▷ tl l :: α list
↳(15) (APP) […] ▷ tl :: α list→α list mit (TL)
↳(15) (…) […] ▷ l :: α list mit (P-ID)

(3) hier wird 2x (P-ID) für exists benutzt:
einmal: exists: (int→bool)→int list→bool; zweimal: exists: (bool→bool)→bool list→bool

Ein Beispiel, bei idem nicht über alle Typvariablen quantifiziert wird:
(1) let f = λx. let g = λy. x in g 0 in not (f 1) ::
↳(2) (P-LET) λx. let g = λy. x in g 0 :: α_1 → α_2
↳(3) (ABSTR) [x: α_1] ▷ let g = λy. x in g 0 :: α_1
↳(4) (P-LET) [x: α_1] ▷ λy. x :: α_2 → α_1
↳(5) (ABSTR) [x: α_1, y: α_2] ▷ x :: α_1 mit (P-ID)
↳(6) (…) […, g: ∀ α_2. α_2→α_1] ▷ g 0 :: α_1
↳(7) (APP) […] ▷ g :: int→α_1 mit (P-ID)
↳(8) (…) […] ▷ 0 :: int mit (INT)
↳(9) (…) […, f: ∀ α_1. α_1→α_1] ▷ not (f 1)
↳(10) (APP) […] ▷ not :: bool→bool | Widerspruch
↳(11) (…) […] ▷ f 1 :: int | nicht wohlgetypt
↳(12) (APP) […] ▷ f :: int→int mit (P-ID)
↳(13) (…) […] ▷ 1 :: int

Beachte: Wenn (P-LET) erlauben würde, über alle Typvariablen zu quantifizieren, dann dürfte man für g in (6) ∀ α_1, α_2. α_2→α_1 eintragen. Also könnte man in (7) z. B. g: int→bool herleiten, was letzten Endes zum Eintrag f: ∀α_1. α_1→bool führen würde, also wäre das Programm wohlgetypt. Aber: Die small step Semantik dieses Programms führt zu „not 1“ ➔ Auswertung bleibt stecken ➔ Typsicherheit wäre verloren.

Überblick über die Sprachen:

	reiner λ-Kalkül	Integers, Booleans, if_then_else	Rekursion	Paare und Listen	
ungetypt	$\mathcal{L}_0$	$\mathcal{L}_1$	$\mathcal{L}_2$	-	Abgeschlossene Ausdrücke können stecken bleiben
explizit getypte Sprache	$\mathcal{L}_0^{t}$	$\mathcal{L}_1^{t}$	$\mathcal{L}_2^{t}$	-	Typsicherheit, d. h. abgeschlossene, wohlgetypte Ausdrücke bleiben nicht stecken
implizit getypt (mit Typinferenz)	$\mathcal{L}_0^{ti}$	$\mathcal{L}_1^{ti}$	$\mathcal{L}_2^{ti}$	$\mathcal{L}_3^{ti}$	
i. g. (mit let-Polymorphie)	$\mathcal{L}_0^{ML}$	$\mathcal{L}_1^{ML}$	$\mathcal{L}_2^{ML}$	$\mathcal{L}_3^{ML}$	

Weiteres Ergebnis:

In der getypten Sprache terminieren alle Programme, die keine explizite Rekursion enthalten (kein „rec“). Das gilt in den ungetypten nicht, z. B. (λx. xx)(λx. xx).