TÍPUSELMÉLET ÉS AZ OBJEKTUM-ELVŰ PROGRAMOZÁS

TYPE THEORY AND OBJECT ORIENTED PROGRAMMING

Csörnyei Zoltán, csz@inf.elte.hu
Eötvös Loránd Tudományegyetem Természettudományi Kar,
Általános Számítástudományi Tanszék

Nagy Sára, saci@inf.elte.hu
Eötvös Loránd Tudományegyetem Természettudományi Kar,
Általános Számítástudományi Tanszék

Abstract

One of the main problems of the implementation of functional programming languages is the correct handling of types. For this the type-theoretic framework is $F_{\leq}^{\omega}$, that is the higher-order polymorphic lambda calculus, which is extended by subtyping and a few extra constructs. The properties of object oriented programming can be described precisely by using this system.

Összefoglaló

A funkcionális programnyelvek implementációjának egyik alapvető problémaköre a típusok helyes kezelése. Az implicit típusos programnyelvekben nem kell a típusokat megadni, a fordítóprogram határozza meg a típusokat a *típuslevezetés* módszereivel; míg az explicit típusos programnyelvek fordítóprogramjai a programokban megadott típusokra *típusellenőrzést* végeznek. A módszerek elméleti háttere a típusos lambda-kalkulus. Ennek absztrakciója az $F_{\leq}^{\omega}$-val jelölt legáltalánosabb típusrendszer, amely a funkcionális programnyelvek fordítóprogramjaiban alkalmazott típuselméleteknek is az alapja. A $F_{\leq}^{\omega}$ típusrendszerrel az objektum elvű paradigma tulajdonságai is egzakt módon leírhatók. Ezekkel a témakörökkel a programtervező matematikus szak I/1. (programnyelvek és automaták) sávjában és az Informatika doktori iskolában foglalkozunk.

TÍPUSELMÉLET ÉS AZ OBJEKTUM-ELVŰ PROGRAMOZÁS

TYPE THEORY AND OBJECT ORIENTED PROGRAMMING

Csörnyei Zoltán, csz@inf.elte.hu
Eötvös Loránd Tudományegyetem Természettudományi Kar,
Általános Számítástudományi Tanszék

Nagy Sára, saci@inf.elte.hu
Eötvös Loránd Tudományegyetem Természettudományi Kar,
Általános Számítástudományi Tanszék

A típusok első formális leírása az 1930-as évek elejére tehető, amikor Haskell B. Curry a lambda-kalkulus egy variánsát, a kombinátor logikát egészítette ki a típus fogalom bevezetésével. A típusfogalom pontos kidolgozása az 1960-as, 70-es években történt meg, amire természetesen már nagy hatással volt a programozás típusfogalma is.

A típus bevezetése nemcsak a programok olvashatóságát, hanem a biztonságosságát is növelte. „A típusos program nem működhet rosszul" – írta R. A. Milner 1978-ban, azt várva, hogy a típusosan helyes programok nem okozhatnak run-time hibát. A típus azóta a programozási nyelvek központi fogalma lett.

A modern típusrendszerek lehetővé teszik azt is, hogy az objektum-elvű programozást formálisan is leírjuk.

A *formális típusrendszer* egy *(S,I,R)* hármas, ahol *S* a típusok leírásának szintaktikáját adja meg, *I* a következtetések formáit írja le, *R* pedig a következtetések érvényességének bizonyítására szolgáló szabályok halmaza.

A gyakorlatban egy programot jól típusozottnak nevezünk, ha a típusellenőrzés fordítási időben nem jelez hibát, és ez azt jelenti, hogy a program futása közben „tilos hiba" biztosan nem fordulhat elő. Ugyanakkor azt mondjuk, hogy egy kifejezés egy típusrendszerben *jól típusozott*, ha a kifejezés típusa egy adott típuskörnyezetben bizonyítható. A két jól típusozott fogalom azonosságát a típushelyesség tételének nevezzük, ezt bizonyítani kell, és egy formális típusrendszer *helyes*, ha benne a típushelyesség tétele bizonyítható.

A típusrendszerek megadásának alapja a típusos lambda-kalkulus, aminek két, szemléletében lényegesen különböző tárgyalási módszere van. Az egyik a *Curry-típusos* leírás, vagy más néven *implicit* típusos rendszer, amelynek az a lényege, hogy a programokban nem kell típust megadni, a programnyelv fordítóprogramja a *típuslevezetés* műveletével határozza meg a típusokat. A másik az *explicit,* vagy *Church-típusos* leírás, ahol a változók típusát meg kell adni, és a kifejezések típusát a fordítóprogram a *típusellenőrzés* folyamán vizsgálja.

1. Típusrendszerek.

1.1. Az elsőrendű típusrendszerek.

Az F_1 típusrendszer típusait az *alaptípusok*, és a típusokból típuskonstrukcióval előállítható függvény típusok adják:

<típus> ::= <alaptípus> | <típus> → <típus> (1)

A következtetések

$\Gamma \vdash$ <típus>, $\Gamma \vdash$ <kifejezés> : <típus> vagy $\Gamma \vdash$ wf (2)

alakúak, ahol wf a „jól formált” környezetet jelenti.

A típusrendszer szabályai közül csak kettőt említünk meg, a függvény absztrakció és az applikáció szabályát:

$$\frac{\Gamma, x:A \vdash E:B}{\Gamma \vdash \lambda x:A.E:A \to B} \qquad \frac{\Gamma \vdash E:A \to B;\Gamma \vdash F:A}{\Gamma \vdash EF:B} \tag{3}$$

Az alaptípusok halmaza a legegyszerűbb rendszerekben a *Bool* és *Nat* típusokból áll, a tipikus bővítések a *Unit*, *Pair*, *Ref*, *List*, *Record* típusokat tartalmazzák.

Az F_1 típusrendszer legfontosabb tulajdonsága az, hogy a redukciók lépéseiben, azaz a program „végrehajtása” folyamán, egy kifejezés típusa nem változhat meg, és egy kifejezés redukálási sorozata biztosan véges, a program biztosan terminál.

Már az F_1 típusrendszerbe bevezethető a reflexív és tranzitív *altípus* fogalom. A *kiterjesztés* szabálya azt mondja ki, hogy ha egy típuskörnyezetben x:A és $A \le B$, akkor x:B is fennáll. Ez a biztonságos helyettesítés elvét adja meg. Az altípus két függvényre is definiálható, az altípus kontravariáns a függvények argumentumára és kovariáns a függvények értékére.

Az objektum-elvű programozás leírásához szükséges az, hogy az altípus fogalmát a *Record* típusra is megadjuk. A rekordokra a „szélességi” és „mélységi” tartalmazással lehet az altípust definiálni:

$$\frac{\Gamma \vdash A_1 \le B_1, \ldots, \Gamma \vdash A_m \le B_m, \Gamma \vdash A_{m+1}, \Gamma \vdash A_n, m \le n}{\{l_1 : A_1, \ldots, l_m : A_m\} \le \{l_1 : A_1, \ldots, l_n : A_n\}}$$

Megjegyezzük, hogy az F_1 rendszerrel definiálható függvények osztálya pontosan azonos a kiterjesztett polinomok osztályával. Ha ezt a rendszert a primitív rekurzióval és egy if-then-else művelettel bővítjük, akkor ebben a bővített rendszerben már például a nem primitív rekurzív Ackermann függvény is leírható.

1.2. Másodrendű típusrendszerek.

Az F_2 típusrendszer az F_2 bővítése. Bevezetjük a *típusváltozó* fogalmát, és a típusváltozókra, mint az egyszerű változókra, típus absztrakciókat és típus applikációkat adunk meg. A típus absztrakciónak is definiáljuk a típusát, és így a *polimorfikus* típusrendszerekhez jutunk el.

Például az *identitás* függvény ebben a rendszerben az

$$Id \equiv \Lambda\alpha.\lambda x{:}\alpha.x \tag{4}$$

alakban írható le, ahol α a típusváltozó, és Λ a típusabsztrakció jele. Az *Id* függvény típusa

$$\forall\alpha.\alpha\rightarrow\alpha. \tag{5}$$

Ha erre applikáljuk például a *Nat* típust, azaz

$$Id\ [Nat] \equiv (\Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha\rightarrow\alpha)\ [Nat] = \lambda x{:}Nat.x : Nat\rightarrow Nat\ , \tag{6}$$

akkor éppen a természetes számokra vonatkozó identitás függvényt kapjuk meg.

1.3. Harmadrendű típusrendszerek.

Ha az F_2 típusrendszerben bevezetett polimorfikus *Id* függvény típusát akarjuk meghatározni, akkor a polimorfikus típusra egy típus absztrakciót bevezetve, azt kapjuk, hogy a függvény típusa

$$\Lambda\omega.\omega\rightarrow\omega, \tag{7}$$

és az *Id* [*Nat*] típusa

$$(\ \Lambda\omega.\omega\rightarrow\omega\ )\ [Nat] = Nat\rightarrow Nat. \tag{8}$$

De mi lehet az ω? Az ω egy „változó”, tetszőleges típus lehet, amit *-gal jelölünk és *kind*-nak, fajtának nevezünk. Így jutunk el az F_3 típusrendszerhez, amelyben

<kind> ::= * | * → <kind> (9)

<típus> ::= <típusváltozó> | <típus> → <típus> | (10)

| ∀ <típusváltozó> : <kind> . <típus>

| Λ <típusváltozó> : <kind> . <típus>

| <típus> <típus>

1.4. Az F^{ω} típusrendszer.

Az F_4 típusrendszerben a kind fogalmát bővítjük,

<kind> ::= * | * → <kind> | <kind> → * , (11)

az F_5 típusrendszerben

<kind> ::= * | * → <kind> | <kind> → * | * * → (* → <kind>), (12)

a további F_6, F_7, … típusrendszerekben a balról és a jobbról zárójelezések minden lehetőségét megadjuk.

Így jutunk el az

$F^{\omega} = F_1 \cup F_2 \cup F_3 \ldots$ (13)

típusrendszerhez, amelyben a

<kind> ::= * | <kind> → <kind> (12)

lesz a <kind> leírása. Ha ezt a rendszert az altípus fogalmával is bővítjük, akkor az $F_{\leq}^{\omega}$ „F-omega-szub” típusrendszert kapjuk meg, az objektum-elvű programozás formális leírására ezt a típusrendszert fogjuk használni.

2. Objektum-elvű modellek

Az objektum-elvű modellek leírásának 3 típusa van:

- a rekord modell. Ez lényegében azt jelenti, hogy az objektumok rekurzív rekordok formájában vannak leírva. A leírás hátránya az, hogy az öröklődés bonyolult rekord konkatenációkkal írható le.

- az egzisztenciális típus modell. Ebben az objektumok egy egzisztenciális típus elemei. A leírás előnye az, hogy a változók konkrét példányai rejtve maradnak. A leírás hátránya az, hogy a metódusok kezelése bonyolulttá válhat.

- az axiomatikus modell. Az objektumokat és metódusokat egy típus nélküli rendszerrel írjuk le, és egy *típusrendszer* szolgál a típusok meghatározására.

A rekord és az egzisztenciális típus modellben az öröklődés alapjai az osztályok, míg a harmadik modellben az öröklődés az objektumokon alapul. Ezekben a rendszerekben a runtime modellek modellezése nehéz, mivel egy új metódust nem lehet egy rekurzív rekord vagy egzisztenciális típus belsejébe behelyezni.

A három modell közül most az egzisztenciális típus modellt mutatjuk be.

3. Az egzisztenciális típus modell

A módszer az $F_{\leq}^{\omega}$ típusrendszeren alapszik. A típus modell első leírása [Pierce 1994]-ben található meg.

3.1. Objektumok és típusaik

Az objektumokat egy egzisztenciális típus elemeiként modellezzük, például a pont objektum típusa a következő:

$$\begin{aligned}\mathsf{Point} = \exists\ \mathsf{Rep}.\{&\mathsf{state} : \mathsf{Rep},\\ &\mathsf{methods}: \{\mathsf{getX}: \mathsf{Rep} \to \mathsf{Int},\\ &\qquad \mathsf{move}: \mathsf{Rep} \to \mathsf{Int} \to \mathsf{Rep}\}\}\end{aligned} \tag{13}$$

Minden objektumnak van egy belső state állapot-komponense és egy metódusokból álló rekordja, amelyek ezen az állapoton operálnak. Az állapot típusa rejtve marad az egzisztenciális kvantor miatt, így a külső függvények ezt nem használhatják.

Az interfész specifikációs függvények típusoperátorok, amelyek egy objektum belső reprezentáció típusát metódus típusok rekordjára képezik le:

$$\begin{aligned}\mathsf{PointM} = \lambda\mathsf{Rep}.\{&\mathsf{getX}: \mathsf{Rep} \to \mathsf{Int},\\ &\mathsf{move}: \mathsf{Rep} \to \mathsf{Int} \to \mathsf{Rep},\\ &\} : \mathsf{Type} \to \mathsf{Type}\end{aligned} \tag{14}$$

Egy pont objektum típusát ezzel a következőképpen írhatjuk le:

$$\begin{aligned}\mathsf{Point} = \exists\ \mathsf{Rep}.\{&\mathsf{state} : \mathsf{Rep},\\ &\mathsf{methods}: \mathsf{PointM}[\mathsf{Rep}]\}\end{aligned} \tag{15}$$

Ebből egy általános objektum típus konstruktort tudunk készíteni:

$$\begin{aligned}\mathsf{Object} = \lambda\mathsf{M} : \mathsf{Type} \to \mathsf{Type}.\ \exists\ \mathsf{Rep}.\{&\mathsf{state} : \mathsf{Rep},\\ &\mathsf{methods}: \mathsf{M}[\mathsf{Rep}]\},\end{aligned} \tag{16}$$

és ezzel

$$\mathsf{Point} = \mathsf{Object}\ \mathsf{PointM}\ . \tag{17}$$

Ha egy p1 pont típusa a fenti Point, akkor az {x: Int} reprezentációs típusú, {x = 1} belső állapotú pont a következőképpen írható le:

$$\begin{aligned}\mathsf{p1} = <\{\mathsf{x}: \mathsf{Int}\}, \{&\mathsf{state} = \{\mathsf{x} = 1\ \},\\ &\mathsf{methods}: \{\mathsf{getX} = \lambda\mathsf{s} : \{\mathsf{x}:\mathsf{Int}\}.(\mathsf{s.x}),\\ &\qquad \mathsf{move} = \lambda\mathsf{s} : \{\mathsf{x}:\mathsf{Int}\}.\lambda\mathsf{i}:\mathsf{Int}.\{\mathsf{x}=\mathsf{s.x}+\mathsf{i}\}\}\} > : \mathsf{Point},\end{aligned} \tag{18}$$

3.2. Üzenetek küldése

Az üzenetküldést polimorfikus függvényekkel modellezhetjük. A polimorfikus függvény paraméterként egy objektumot kap, azt „megnyitva" alkalmazzuk a megfelelő metódust, majd „összecsomagoljuk". A move üzenet elküldésére a következő függvényt használhatjuk:

```
Point'move = λM ≤ PointM. λp : Object M.                                    (19)
             open p as <Rep,r> in
               λd: Int. < Rep,
                        {state = r.methods.move(r.state) d,
                         methods = r.methods}
                       >: Object M
             end;
```

Ezzel a

```
Point'move [Object PointM] p1 2                                             (20)
```

kifejezés ad egy x = 3 állapotú pontot.

3.3. Öröklődés, egységbe foglalás és további jellemzők

Az objektum elvű programok formális leírásának további tulajdonságai is hasonlóan egyszerű módon leírhatók.

4. Irodalomjegyzék

[Abadi 1996]
Abadi, M. and Cardelli, L. „A Theory of Objects", Springer-Verlag, 1996.

[Cardelli 1985]
Cardelli, L. and Wegner, P. „On understanding types, data abstraction, and polymorphism", Computing Surveys, 17(4):471-522, 1985.

[Fischer 1996]
Fisher, K. and Mitchell, J. C. „The Development of Type Systems for Object-oriented Languages", Theory and Practice of Object Systems 1(3):189-220, 1996.

[Pierce 1994]
Pierce, B.C. and Turner, D.N. „Simple type-theoretic fundations for object-oriented programming", Journal of Functional Programming, 4(2):207-248., 1994.